제3장 C 프로그래밍 환경

학습 목표

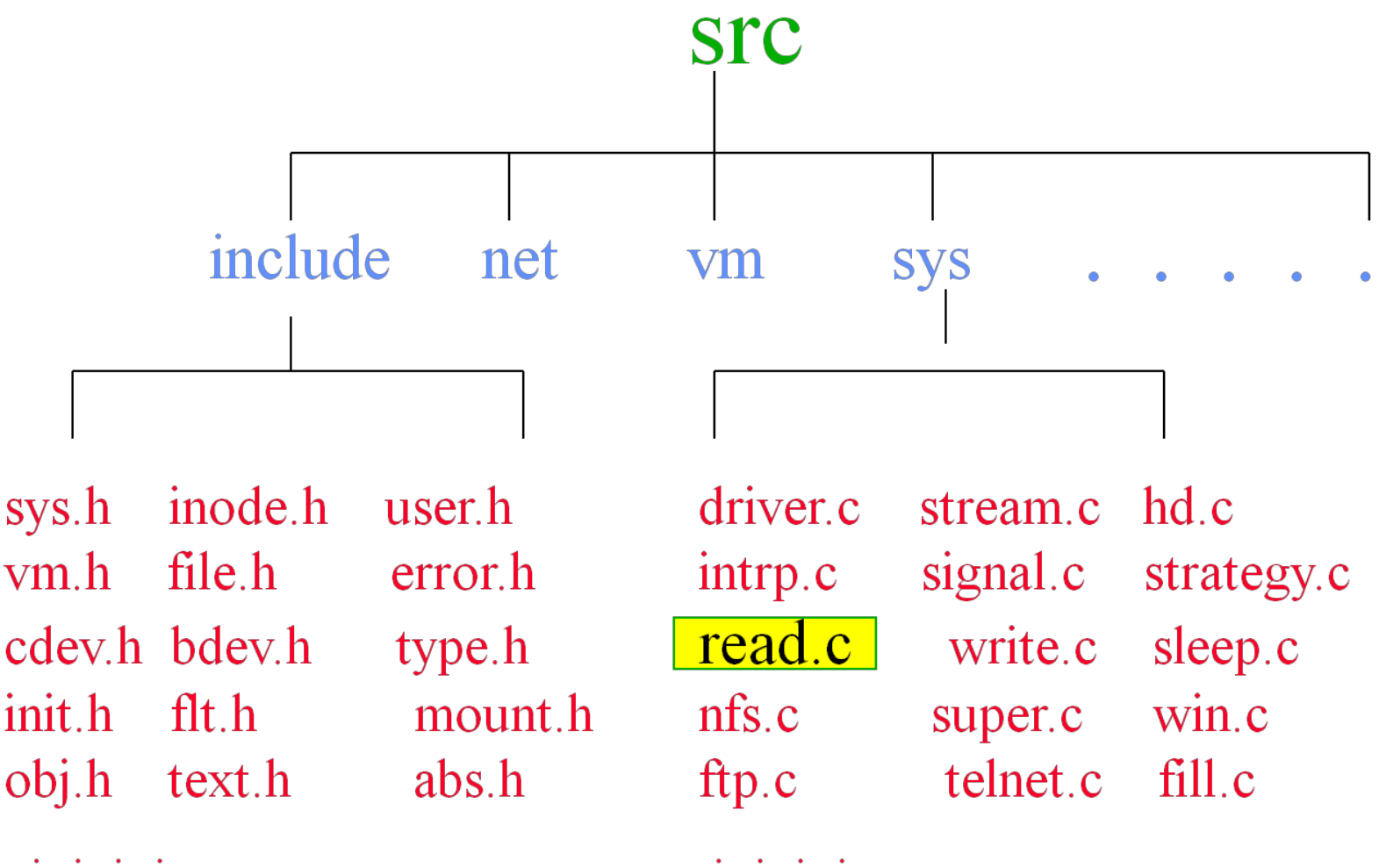
- 문서 편집 : vi, geidt
- C 컴파일러 사용: gcc
- 컴파일 자동화: make
- 디버깅: gdb
- 통합개발환경: Eclipse
- 라이브러리 관리: ar
- 소스 관리: ctags
- 형상 관리: CVS, SVN, git

3.2 make 시스템

make 시스템

- make 시스템
  - 대규모 프로그램의 경우에는 헤더, 소스 파일, 목적 파일, 실행 파일의 모든 관계를 기억하고 체계적으로 관리하는 것이 필요
  - 파일들의 의존관계를 파악
  - make 시스템을 이용하여 효과적으로 작업
  - 필요한 파일만 다시 컴파일하여 실행파일을 재생성
- Makefile
  - 실행 파일을 만들기 위해 필요한 파일들과 만드는 방법을 기술
  - make 시스템은 파일의 상호 의존 관계를 파악하여 실행 파일을 쉽게 다시 생성
- $ make [-f 메이크파일]
  - 옵션이 없으면 기본 make 파일인 Makefile 혹은 makefile을 사용

make 시스템

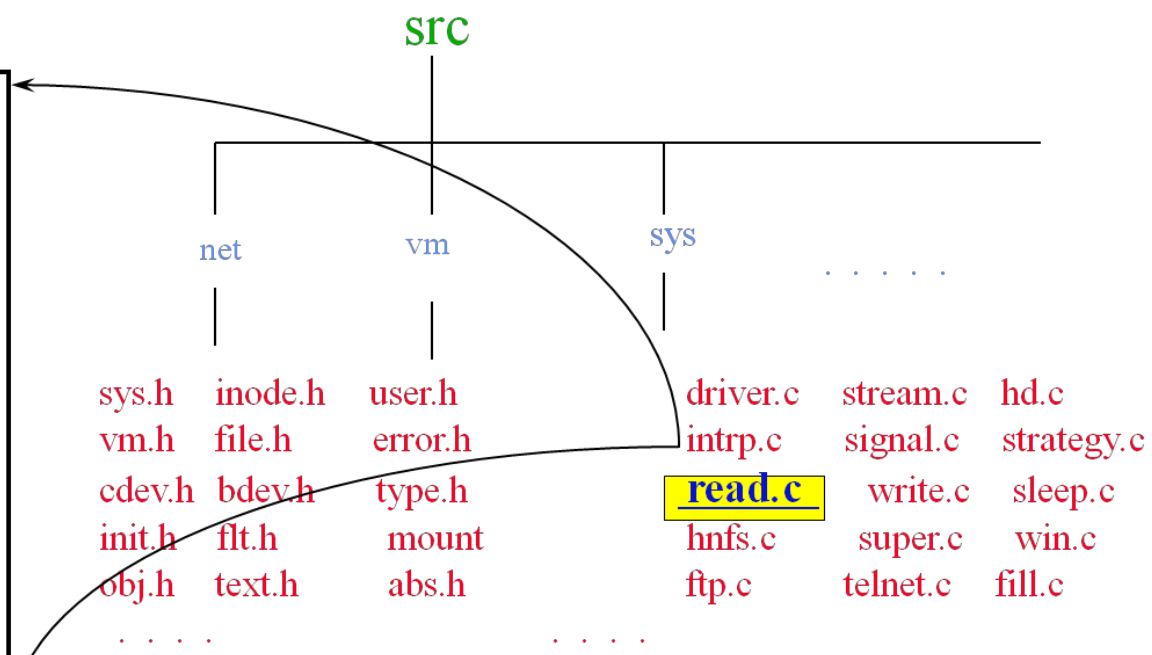

src
include
net
vm
sys
. . . . .
sys.h
vm.h
cdev.h
init.h
obj.h
. . . .
inode.h
file.h
bdev.h
flt.h
text.h
user.h
error.h
type.h
mount.h
abs.h
driver.c
intrp.c
read.c
nfs.c
ftp.c
. . . .
stream.c
signal.c
write.c
super.c
telnet.c
hd.c
strategy.c
sleep.c
win.c
fill.c

make 시스템

Reading and Modifying Codes . . read.c

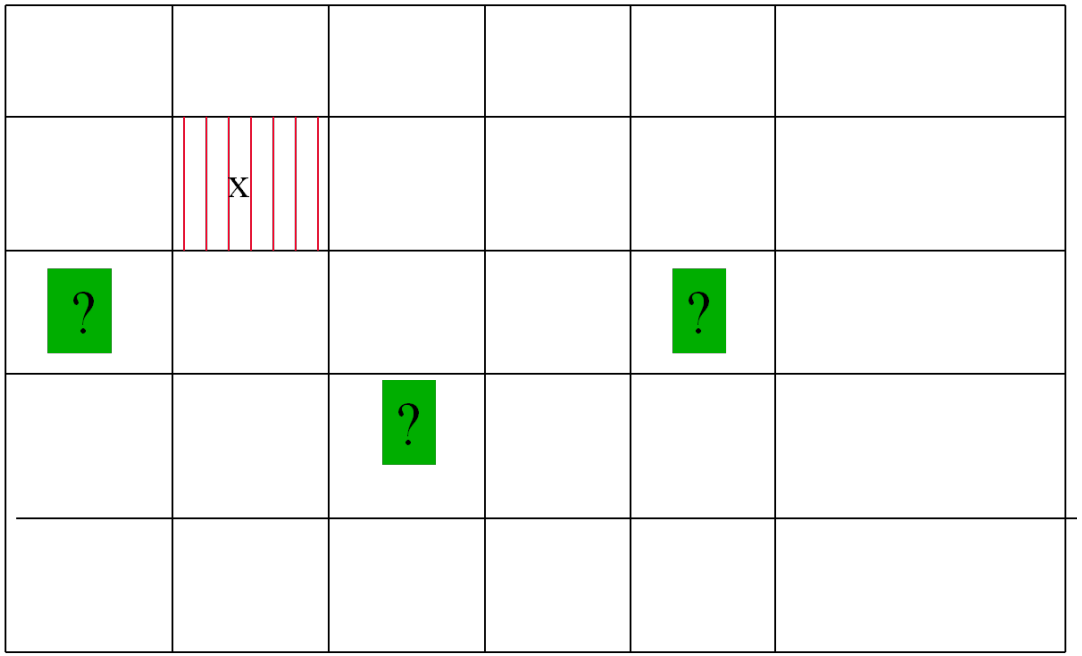
```
#include        "user.h"
#include        "sys.h"
#include        "type.h"

int read (pst)
struct      buf       *pst;
{
char      ps[MAX];
int      index;
         while  (i ... )
         {   ...    }
```



make 시스템 컴파일 자동화

After changing a source code file?



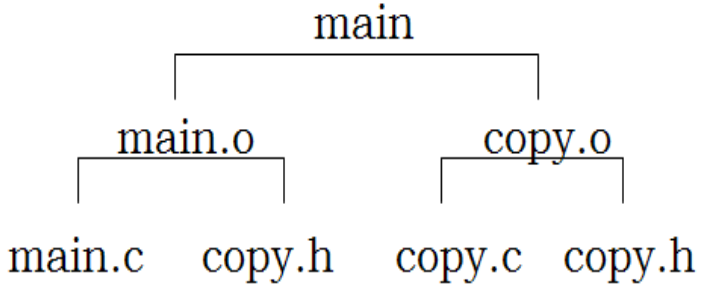
1. Modify x in A.c
2. Compile only A .c? Need something else?

Makefile의 구성

- Makefile의 구성 형식
  대상리스트: 의존리스트
  (tab)명령리스트

- 예: Makefile

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



Makefile의 구성

- make 실행

  $ make 혹은 $ make main

  gcc -c main.c

  gcc -c copy.c

  gcc -o main main.o copy.o

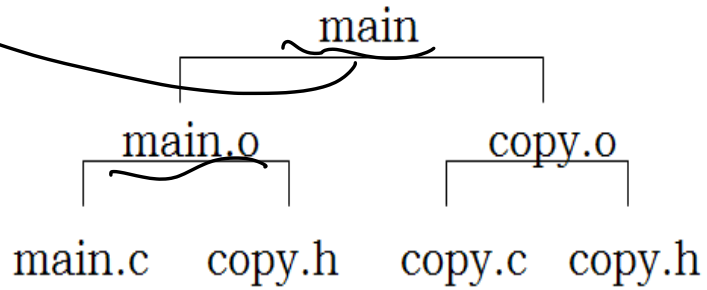
- copy.c 파일이 변경된 후

  $ make

  gcc -c copy.c

  gcc -o main main.o copy.o

  → 이 경우 main.c의 재컴파일 없음



Label

- 레이블
  - 레이블로 사용될 때는 의존 관계 부분은 없어도 된다.
  - 즉, 명령어의 수행만 필요할 경우에 사용

- Makefile 내부의 label 사례
  - clean :

    rm -f *.o

- %> make clean (레이블 호출)

  rm main.o read.o write.o 가 실행됨

Label이 추가된 makefile

```
[cse_yu]$ more Makefile
maketest : main.o sub1.o sub2.o
            gcc -o maketest main.o sub1.o sub2.o
main.o : main.c io.h
            gcc -c main.c
sub1.o :  sub1.c io.h
            gcc -c sub1.c
sub2.o : sub2.c io.h
            gcc -c sub2.c
clean :
            rm -f *.o maketest
```

실행

$ make

→ make 후 .o 파일을 삭제할 경우,

$ rm *.o or $ make clean

Make 예제

ex8.c

```
#include <stdio.h>
int main(int argc, char **argv)
{
    printf("Hello World!₩n");
    return 0;
}
```

=makefile

Makefile

```
ex8: ex8.c
    gcc -o ex8 ex8.c
```

```
[sugar@cse_yu]$ make
gcc -o ex8 ex8.c
[sugar@cse_yu]$ make
make: `ex8' is up to date.
[sugar@cse_yu]$ touch ex8.c
[sugar@cse_yu]$ make
gcc -o ex8 ex8.c
[sugar@cse_yu]$ make
make: `ex8' is up to date.
```

접근시간 바뀜

→ 소스의 일관성을 유지시켜 줌

→ vi로 ex8.c내용을 갱신

```
[sugar@cse_yu]$ make
gcc -o ex8 ex8.c
```

***touch*는 파일의 수정 시간을 현재시간으로 수정**

Makefile 파일

- Make 유틸리티의 기본 입력화일
  - Makefile (or makefile)
  - 단, 하나의 디렉토리에는 하나의 Makefile만이 존재해야 함

- Makefile 파일명 외의 다른 파일명을 갖는 입력 파일 사용
  - make -f <입력파일명>
    - 임의의 이름으로 makefile을 작성할 수 있게 함.

  - 예) make -f mymakefile.ex9

Label 및 target 호출 (요약)

구분	명령어 실행
◆ **Makefile(makefile)**이 있을 때	**$make**
- 임의의 타겟 호출	**$make target_name**
- 임의의 레이블 호출	**$make label_name**
◆ 임의의 파일로 있을 때**(-f** 옵션 필요**)**	**$make –f file_name**
- 임의의 타겟 호출	**$make –f file_name target_name**
- 임의의 레이블 호출	**$make –f file_name label_name**

Macro

- 매크로
  - Makefile에 기술되는 반복적인 내용을 단순화 시키기 위해 사용
  - 매크로 정의는 "="를 포함하는 하나의 문장
  - "#"는 주석문의 시작을 의미
  - 여러 행을 사용할 때에는 "₩"를 사용
  - 매크로 참조의 경우 "$" 후, 소괄호나 중괄호를 사용

정의: macro_name=text_string
참조: ${macro_name}

Macro

- 종류
  - 사용자 정의 매크로 (user defined macro)
  - 사전 정의 매크로 (predefined macro)
  - 내부 매크로 혹은 자동 매크로 (auto macro)

- 기타
  - 정의되지 않은 매크로는 null 문자로 치환
  - 중복된 정의는 가장 마지막 정의값이 사용

```
NAME = stringA
NAME = stringB
${NAME}          → 이 경우 stringB로 치환
```

  - 매크로 정의 시에 이전에 정의된 매크로 사용 가능

```
NAME = string
NAME2 = my ${NAME}      → NAME2에는 my string이 정의됨
```

User Defined Macro

- 반복적으로 사용되는 내용을 C언어의 #define과 같이 사용
- 임의의 매크로 변수에 사용이 가능

```
maketest : main.o sub1.o sub2.o
        gcc –o maketest main.o sub1.o sub2.o
```

```
OBJECTS= main.o sub1.o sub2.o

maketest : ${OBJECTS}
        gcc –o maketest ${OBJECTS}
```

Predefined Macro

- 내부적으로 미리 정의되어 있는 매크로의 사용
  - Make 유틸리티가 인식할 수 있도록 사전 정의한 매크로 및 환경변수
  - 정확히는, 정의 대상이 미리 약속된 매크로
- 확인 : make -p
  - make 에서 미리 세팅되어 있던 모든 값들 확인
    - $(CC) 항상 C 컴파일러로 인식
    - $(LD) 항상 로더(링커)로 인식

```
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
```

Predefined Macro

매크로	내용
CFLAGS	cc와 gcc의 옵션을 지정한다.
CC	C 컴파일러를 지정한다. (CC=gcc 혹은 CC=cc)
ASFLAGS	as의 옵션을 지정한다.
AS	어셈블러를 지정한다. (AS=as)
CPPFLAGS	c++와 g++의 옵션을 지정한다.
CXX	C++ 컴파일러를 지정한다. (CXX=g++)
LDFLAGS	ld의 옵션을 지정한다.
LD	ld 프로세스를 지정한다. (LD=ld)
LFLAGS	lex의 옵션을 지정한다.
LEX	lex 프로세스를 지정한다. (LEX=lex)
YFLAGS	yacc의 옵션을 지정한다.
YACC	yacc 프로세스를 지정한다. (YACC=yacc)

사용자 정의 및 사전 정의 매크로 사용 예 1

```
maketest : sub1.o sub2.o main.o
    gcc -o maintest main.o sub1.o sub2.o
```

```
OBJECTS = sub1.o sub2.o main.o
TARGET = maketest
CC=gcc
CFLAGS=-c
${TARGET} : ${OBJECTS}
        ${CC} -o ${TARGET} ${OBJECTS}
main.o : main.c
        ${CC} ${CFLAGS} main.c
sub1.o :  sub1.c
        ${CC} ${CFLAGS} sub1.c
sub2.o : sub2.c
        ${CC} ${CFLAGS} sub2.c
clean :
        rm -f ${OBJECTS} ${TARGET}
```

사용자 정의 및 사전 정의 매크로 사용 예 2

ex9-a.c

```
#include <stdio.h>
void a(void)
{
    printf("func a₩n");
}
```

ex9-b.c

```
#include <stdio.h>
void a(void);
int main(int argc, char **argv)
{
    a();
    return 0;
}
```

Makefile.ex9

```
CC = gcc
SRC = ex9-a.c ex9-b.c
ex9 : ${SRC}
   ${CC} ${SRC} -o ex9
```

```
[sugar@cse_yu]$ make -f Makefile.ex9
gcc ex9-a.c ex9-b.c -o ex9
[sugar@cse_yu]$ ./ex9
func a
[sugar@cse_yu]$
```

Auto Macro

- 자동 매크로 혹은 내부 매크로
  - 내부적으로 정의
  - make –p 명령어로 확인 할 수 없는 매크로들
  - 특별한 의미의 매크로이며 의미의 수정이 불가능

Auto Macro

- $*
  - 의존 관계가 있는 파일 중 현재 처리 중인 파일의 확장자를 제외한 이름
  - 즉, 확장자가 없는 현재 목표 파일의 이름을 지칭
  - 확장자 규칙에서 사용
  - "$*.c" 의 형태로 사용
- $<
  - 의존관계가 있는 파일 중 현재 처리 중인 파일의 이름
  - 즉, 일반적으로 첫번째 종속물 (first prerequisite), 즉 소스 파일을 의미
  - 현재 타겟 파일보다 더 최근에 업데이트한 파일명
- $@
  - 현재 목표(target) 파일 명

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

```
OBJS=main.o sub1.o sub2.o
maketest : $(OBJS)
    gcc -o $@ ${OBJS}   ← gcc -o maketest  $(OBJS)
```

```
.c.o :   ← .c는 .o로 변환되어야 한다는 의미
gcc -c $< (또는 gcc -c $*.c)  ← gcc -c main.c
```

확장자 규칙(suffix rule)

- .SUFFIX 매크로
  - 확장자 규칙이란 파일의 확장자를 보고, 그에 따라 적절한 연산을 수행시키는 규칙
  - make 파일에게 주의 깊게 처리할 파일들의 확장자를 등록
  - 오브젝트 파일이 존재하지 않으면, make는 이를 생성하기 위해 .c, .s 소스 파일을 찾음
  - 확장자 규칙에 의해 make는 파일들간의 확장자를 자동으로 인식해서 필요한 작업을 수행

  - .c .o
    - .c를 확장자 파일을 .o를 확장자로 갖는 파일로 바꾸라는 규칙

  - 사용 예
    - .SUFFIXES : .c .o
    - 이렇게 선언하면, make 내부에서는 미리 정의된 .c (C 소스 파일)를 컴파일해서 .o (목적 파일)를 만들어 내는 루틴이 자동적으로 동작

확장자 규칙

다음의 세가지 형태의 makefile은 같은 기능을 수행 한다.

1. 기본 Makefile

```
OBJECTS = sub1.o sub2.o main.o
TARGET = maketest

${TARGET} : ${OBJECTS}
    gcc -o $@ ${OBJECTS}

main.o : main.c io.h
    gcc -c main.c
sub1.o :  sub1.c io.h
    gcc -c sub1.c
sub2.o : sub2.c io.h
    gcc -c sub2.c

clean :
    rm -f ${OBJECTS} ${TARGET}
```

2. Default 확장자 규칙

```
OBJECTS = sub1.o sub2.o main.o
TARGET = maketest

${TARGET} : ${OBJECTS}
    gcc -o $@ ${OBJECTS}

main.o : main.c io.h
sub1.o :  sub1.c io.h
sub2.o : sub2.c io.h

clean :
    rm -f ${OBJECTS} ${TARGET}
```

→Make가 이미 알고 있는 Command list는 생략 가능 (e.g., gcc –c man.c)
이를 명시적으로 지정 → 우측 사례
suffixes

→ 생략되어 있는 CFLAGS 혹은 CC 매크로는 default 값을 가질 것임을 의미.
이를 명시적으로 지정하는 방식 → 우측 사례

3. 명시적으로 확장자 지정

```
.SUFFIXES:.c.o    //확장자 규칙 선언 .c가 .o로  변환

OBJECTS = sub1.o sub2.o main.o
TARGET = maketest
CFLAGS=-c
CC=gcc          //매크로 명시적 재정의

${TARGET} : ${OBJECTS}
    ${CC} -o $@ ${OBJECTS}
.SUFFIXES: .c .o
.c.o:   //확장자 규칙 적용시 사용 명령어 명시적 지정
    ${CC} ${CFLAGS} $< -o $@

clean :
    rm -f ${OBJECTS} ${TARGET}

main.o : main.c io.h
sub1.o :  sub1.c io.h
sub2.o : sub2.c io.h   //명령리스트 생략
```

target

기타, 자동 의존 관계 생성

- gccmakedep
  - 어떤 파일의 의존 관계를 자동으로 조사해서 Makefile 의 뒷부분에 붙여주는 유틸리티

- **의존성 추가 없이:** `math.h`가 변경되어도 `main.o`가 재컴파일되지 않음.
- **의존성 추가 후:** `math.h`가 변경되면 `main.o`와 `math.o` 모두 재컴파일됨.

math.h가 makefile에 사용이 안될때

- gcc –M *.c
  - gccmakedep과 같은 역할 수행

- 사용 예
  - make dep

```
.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)
```

여러 목표 프로그램

- Make 파일 내부에 목표 프로그램이 다수 일 경우
  - Make 명령어는 기본적으로 가장 상위 목표 프로그램만 처리
  - 상호 의존 관계가 없는 두 목표 프로그램의 처리 시, target을 모두 지정해야 함
    - 혹은 별도의 makefile로 작성하여 관리

여러 목표 프로그램

파일 지정 안해주면
Makefile의 첫 번재 타겟이 실행

Makefile.ex8-9

```
ex8: ex8.c
    gcc ex8.c -o $@

CC = gcc
SRC = ex9-a.c ex9-b.c

ex9: ${SRC}
    ${CC} ${SRC} -o $@
```

```
[sugar@cse_yu]$ make -f Makefile.ex8-9
gcc -o ex8 ex8.c     ← ex8만 실행 됨
[sugar@cse_yu]$ rm ex8.o ex9-a.o ex9-b.o ex9 ex8

[sugar@cse_yu]$ make -f Makefile.ex8-9 ex8 ex9
gcc ex8.c -o ex8
gcc ex9-a.c ex9-b.c -o ex9
[sugar@cse_yu]$
```

여러 목표 프로그램의 개선

→ Makefile.ex8-9의 첫줄에 추가
all: ex8 ex9

→ 마지막 줄에 다음을 추가.
clean:
	rm ex8.o ex9-a.o ex9-b.o ex9 ex8

[sugar@cse_yu]$ make -f Makefile.ex8-9 clean
[sugar@cse_yu]$ make all
.
. //make 실행
.
[sugar@cse_yu]$ make clean

여러 목표 프로그램의 예제

```
.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
```

← 다중 타켓

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

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

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

dep :
        gccmakedep $(SRCS)
```