프로그램 개발 도구
Program Development Tools

- gcc
- gdb
- make

프로그램 개발 환경

- 필수 도구
  - 소스 편집기: vi, emacs, …
  - 컴파일러: gcc
  - 실행환경: shell
  - 디버거: gdb
- 그 외 유용한 도구
  - 프로젝트 관리: make, autoconf, automake
  - 버전 관리: cvs, svn
  - 패키지 관리: tgz, rpm, dpkg(deb), …
- Integrated Development Environment (IDE)
  - GUI 환경에서 동작
  - 편집기, 컴파일러, 실행 환경, 디버거 등이 모두 통합되어 있음
  - Eclipse, KDevelop, anjuta, …

IDE

- ❑ KDevelop
  - C/C++을 비롯한 많은 언어의 개발 환경
  - Qt/KDE 기반 GUI 프로그램 가능
  - Linux, MS Windows, Mac OsX, Solaris 등 다양한 플랫폼에 설치 가능
- ❑ Eclipse
  - Java를 비롯한 많은 언어의 개발 환경
  - Multi-platform
  - Plug-in 기능을 이용한 확장성 제공
- ❑ QDevelop
  - Qt4에 기반한 cross-platform 개발 환경
  - Multi-platform

IDE

- ❑ KDevelop 설치
  - 프로그램 설치/제거 메뉴 활용

IDE

❑ KDevelop 설치

- 앞의 과정만으로 프로그램 개발이 되지 않을 때…
  - 관련 도구 미비
  - libtool 버전 불일치
- 권장 설치 과정
  - 개발도구 설치: sudo apt-get install build-essential
  - KDevelop 설치: sudo apt-get install kdevelop
  - konsole 설치: sudo apt-get install konsole
  - libtool downgrade 또는 configuration 파일 수정
    - libtool downgrade: libtool_1.5.26-1ubuntu1_i386.deb 설치 (http://doc.integrasoftware.it/tec/sis/pub/ubuntu904kdevlibtool)
    - Configuration 파일 수정 (게시판 참조)

IDE

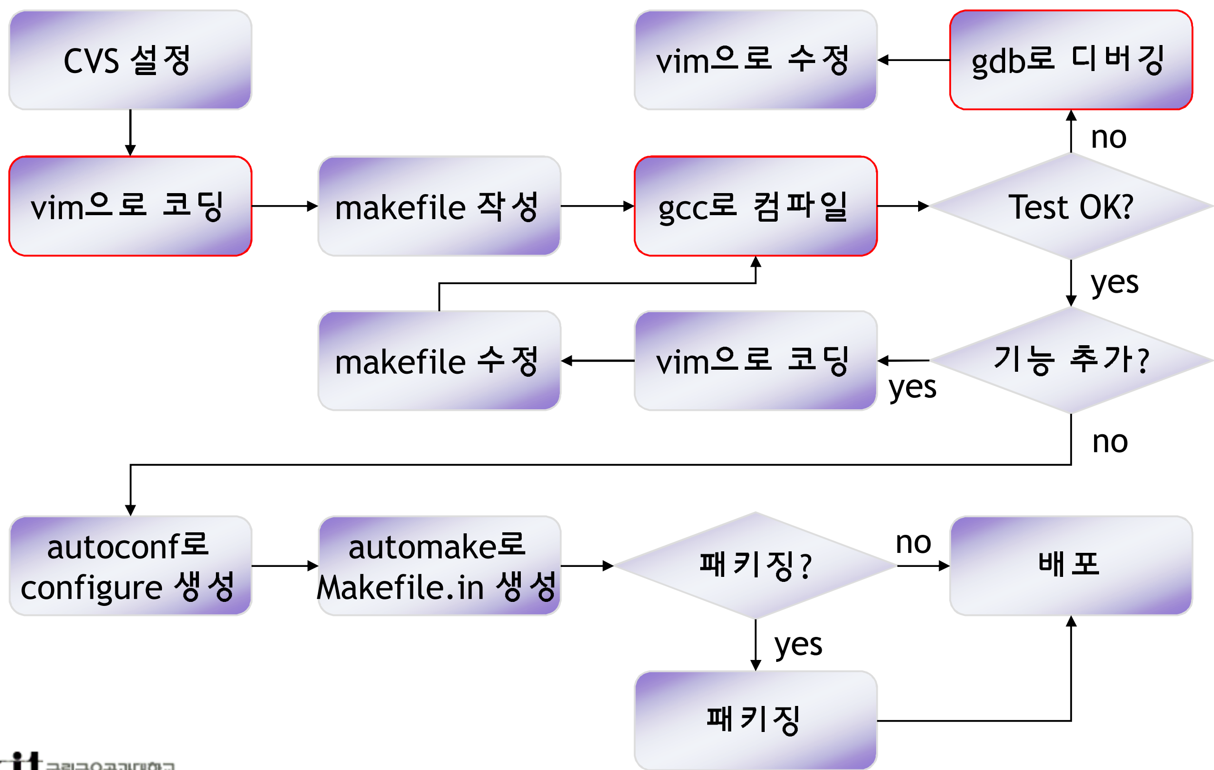
❑ KDevelop을 이용한 프로그램 개발

- Project 생성 → 소스 편집 → 컴파일/실행
- Debug 기능
- 단순한 콘솔 프로그램에는 오버헤드 과다

❑ 사용법 (스스로 공부합시다)

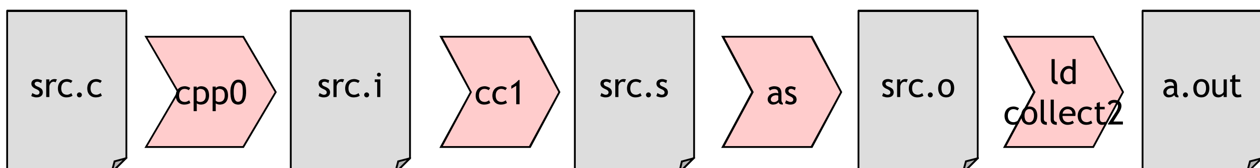
- 튜토리얼: http://www.kdevelop.org/

프로그램 개발 과정



gcc

- 리눅스 표준 컴파일러
  - GNU에서 개발
- gcc는 C 컴파일러가 아니다!
  - 전처리기(preprocessor), 컴파일러, 어셈블러, 링커 등을 모두 종합한 도구
  - C, C++, Fortran, Ada, Objective-C 등 다른 언어 컴파일도 가능



gcc

❑ gcc 수행 과정

- 전처리 (preprocessing)
  - #으로 시작하는 전처리 명령을 처리하여 소스 파일을 변경한다.
- 컴파일 (compile)
  - 프로그램을 문법과 약속된 의미에 따라 분석하여 어셈블리어로 번역
  - 최적화(optimization)
- 어셈블 (assemble)
  - 어셈블리어로 된 프로그램을 기계어(binary) 파일로 변환
  - ELF (Executable and Linking Format) 헤더 첨부
- 링크 (link)
  - 라이브러리 연결

gcc

❑ 확인해 보자!

```
$ gcc –v --save-temps –o src src.c
…
$ gcc –c src.c
$ ls
   a.out  src.c  src.i  src.o  src.s
$ vi src.i (또는  less src.i)
$ vi src.s (또는 less src.s)
$ objdump –S src.o
$ objdump –S src
$ readelf –a src.o
$ readelf –a src
```

gcc

❑ gcc 사용법

```
$ gcc [options] source_files [post-options]
```

- 옵션을 사용한 컴파일 예

```
$ gcc –v –I/usr/local/include –DDEBUG –Wall –W –O2 –L/usr/local/lib –o prgm prgm.c -lm
```

gcc 옵션 (–v), cpp0 옵션 (–I/usr/local/include –DDEBUG), cc1 옵션 (–Wall –W –O2), collect2 옵션 (–L/usr/local/lib –o prgm, -lm)

- 가장 간단한 사용

```
$ gcc prgm.c
```

  - 컴파일 전체 과정을 수행한 후 실행파일로 a.out을 만든다.

gcc

❑ 주요 옵션

옵션	의 미
-o *filename*	a.out 대신 *filename* 실행 파일 생성
-c	어셈블까지만 수행 (*.o 파일 생성)
-S	컴파일까지만 수행 (*.s 파일 생성)
-v	컴파일 과정의 메시지를 화면으로 출력
--save-temps	중간에 생성되는 임시 파일을 삭제하지 않고 저장
-I*dir*	헤더 파일 탐색 경로에 *dir* 추가 (/usr/include, /usr/local/include, /usr/lib/gcc-lib/i386-readhat-linux/3.2.2/include 등은 기본)
-D*macro*	macro를 외부에서 정의
-M, -MM	종속 항목들을 출력

gcc

❑ 주요 옵션

옵션	의 미
-W	합법적이지만 모호한 코딩에 대한 경고
-Wall	모든 모호한 코딩에 대한 경고
-O*n*	최적화. 숫자 n은 0~3의 정수로 클수록 높은 효율의 코드 생성
-g	gdb(디버거)에게 제공하는 정보 삽입
-L*dir*	라이브러리 탐색 경로에 *dir* 추가 (/usr/lib, /usr/lib/gcc-lib/i386-readhat-linux/3.2.2가 기본)
-l*lib*	라이브러리 *lib*을 링크 라이브러리 이름 libxxx.* → xxx만 쓴다.

gcc

❑ gcc 사용 예

juyoon@ce:~/system/programming

```
[juyoon:programming/15]$ ls -R
.:
합계 8
   4 include/      4 src/

./include:
합계 4
   4 test.h

./src:
합계 8
   4 print.c      4 test.c
[juyoon:programming/16]$
```

juyoon@ce:~/system/programming

```
#define MAX 100

extern void print (double *);
~
~
~
~
~
"include/test.h" 3L, 47C          3,1
```

juyoon@ce:~/system/programming

```
#include <math.h>
#include "test.h"

int main (void)
{
    double arr[MAX];
    int i;

    for (i=0; i<MAX; i++)
        arr[i] = sqrt((double) i);
    print(arr);

    return 0;
}
"src/test.c" 14L, 162C          13,2-5          모두
```

juyoon@ce:~/system/programming

```
#include <stdio.h>
#include "test.h"

void print (double *a)
{
    int i;

    for (i=0; i<MAX; ++i)
        printf("sqrt(%d) = %lf\n", i, a[i]);
}
~
~
~
~
"src/print.c" 10L, 136C          9,3-9          모두
```

gcc

❑ gcc 사용 예

```
[juyoon:programming/43]$ gcc -o t src/test.c
src/test.c:2:18: test.h: 그런 파일이나 디렉토리가 없음
src/test.c: In function `main':
src/test.c:6: `MAX' undeclared (first use in this function)
src/test.c:6: (Each undeclared identifier is reported only o
nce
src/test.c:6: for each function it appears in.)
[juyoon:programming/44]$
```

```
[juyoon:programming/45]$ gcc -o t -Iinclude src/test.c
/tmp/cc8aD2y0.o(.text+0x40): In function `main':
: undefined reference to `sqrt'
/tmp/cc8aD2y0.o(.text+0x63): In function `main':
: undefined reference to `print'
collect2: ld returned 1 exit status
[juyoon:programming/46]$
```


gcc

❑ gcc 사용 예

```
[juyoon:programming/47]$ gcc -o t -Iinclude src/test.c src/print.c
/tmp/ccWzKKB9.o(.text+0x40): In function `main':
: undefined reference to `sqrt'
collect2: ld returned 1 exit status
[juyoon:programming/48]$
```

```
[juyoon:programming/49]$ gcc -o t -Iinclude src/test.c src/print.c -lm
[juyoon:programming/50]$ t | more
sqrt(0) = 0.000000
sqrt(1) = 1.000000
sqrt(2) = 1.414214
sqrt(3) = 1.732051
sqrt(4) = 2.000000
sqrt(5) = 2.236068
sqrt(6) = 2.449490
sqrt(7) = 2.645751
sqrt(8) = 2.828427
sqrt(9) = 3.000000
```

gdb

❑ 디버깅

- 프로그램 오류를 수정하는 과정
- design, syntax, semantics, run-time 등 다양한 단계에서 오류 발생
  - 컴파일러는 syntax 오류와 일부 semantics 오류를 검사해 준다.
- 논리 오류의 경우 다양한 입력에 대해 일일이 실행 과정을 추적하며 찾아내야 함.
  → 디버거 사용으로 명랑 프로그래밍 사회 구현!

❑ gdb (GNU Project Debugger)

- GNU에서 구현한 "강력" 한 디버거
- http://www.gnu.org/software/gdb/documentation
- http://korea.gnu.org/manual/release/gdb

gdb

❑ 컴파일 옵션

- -g 옵션을 주고 컴파일해야 gdb를 사용할 수 있다.
- 디버깅할 때는 -O 옵션을 주지 않는다.

❑ gdb 시작

```
$ gdb [options] [program] [core file]
$ gdb [options] [program] [process ID]
```

- core: 특정 오류에 대해 OS가 생성하는 파일
  - 종료 순간의 CPU context와 데이터, 스택 등의 정보 저장
- process ID
  - 실행 중인 프로그램의 디버깅

gdb

❑ gdb 종료

- "q"(quit) 또는 "^D" 입력

```
juyoon@ce:~/system/programming/debug
[juyoon:debug/7]$ gdb buggy
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) q
[juyoon:debug/8]$ gdb buggy
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) [juyoon:debug/9]$
```


gdb

❑ gdb 명령

- 주로 약자 사용
  - 완성하고 싶으면 TAB 입력
- h (help) - 도움말 출력
  - 단계적으로 되어 있어 "help [group]" 또는 "help [command]"로 상세 정보를 볼 수 있다.
- l (list) - 소스 보기
  - main 함수를 기점으로 실행 가능 코드부터 10줄씩 출력
  - l *n*   *n*번째 행을 기준으로 출력
  - l *func*   함수 *func*의 소스 출력
  - l -   이전 10줄을 출력
  - l file.c:func   file.c 파일의 func 함수를 출력
  - l file.c:n   file.c 파일의 n번째 행을 기준으로 출력
  - 출력되는 행의 수를 변경하고 싶으면: set listsize 20

gdb

❑ gdb 명령

- 프로그램 실행 명령
  - r (run) - 프로그램 실행
    - 별다른 설정이 없으면 종료할 때까지 실행
    - r [args] : 명령행 인자가 있는 경우
  - k (kill) - 실행 중인 프로그램 강제 종료
  - bt (backtrace) - 종료 시 스택 상태를 보여 줌

```
(gdb) r
Starting program: /home/juyoon/system/programming/debug/buggy
array[i] = one
array[i] = two
array[i] = three
array[i] = (null)

Program received signal SIGSEGV, Segmentation fault.
0x4207a42b in strlen () from /lib/tls/libc.so.6
(gdb) bt
#0  0x4207a42b in strlen () from /lib/tls/libc.so.6
#1  0x4204752d in vfprintf () from /lib/tls/libc.so.6
#2  0x4204f112 in printf () from /lib/tls/libc.so.6
#3  0x080483bc in main () at buggy.c:32
#4  0x42015574 in __libc_start_main () from /lib/tls/libc.so.6
(gdb)
```


gdb

❑ gdb 명령

- Breakpoint 명령
  - b (breakpoint) - 실행 중단 지점 설정

옵션	의 미
func	func 함수 시작 부분에 breakpoint (bp) 설정
10	10행에 bp 설정
file.c:func	file.c 파일의 func 함수에 bp 설정
file.c:10	file.c 파일의 10행에 bp 설정
+2, -2	현재 행에서 2행 다음(전)에 bp 설정
10 if var==0	10행에 bp를 설정하는데 var의 값이 0일 때만 작동

  - cl (clear) - breakpoint 지우기
    - 옵션은 설정 때와 같다.
  - d (delete) - 모든 breakpoint 지우기

gdb

❑ gdb 명령

- 단계별 진행 명령
  - breakpoint로 중단된 지점부터의 실행 방식 제어
  - s (step)
    - 한 행을 수행한 후 정지
    - 함수 호출 시 함수 내부로 들어간다.
    - 숫자를 붙이면 그 수만큼 반복
  - n (next)
    - 한 행을 수행한 후 정지
    - 함수 호출 시 함수 실행 후 다음 행으로 진행
    - 숫자를 붙이면 그 수만큼 반복
  - c (continue)
    - 다음 breakpoint 만날 때까지 계속 진행

gdb

❑ gdb 명령

- 단계별 진행 명령 (계속)
  - u (until) - 현재 loop를 빠져 나감
  - finish - 현재 함수를 수행하고 나감
  - return - 현재 함수를 수행하지 않고 빠져 나감
    - return *value*: 현재 함수를 수행하지 않으나 반환값으로 *value*를 돌려 줌
  - watch *var* - 변수 *var*의 값이 변할 때마다 정지
    - 실행 중인 경우에만 사용 가능

gdb

❑ 단계별 실행 예

```
juyoon@ce:~/system/programming/debug
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) b main
Breakpoint 1 at 0x8048381: file buggy.c, line 27.
(gdb) r
Starting program: /home/juyoon/system/programming/debug/buggy

Breakpoint 1, main () at buggy.c:27
27              int lval = 2331, i;
(gdb) watch i
Hardware watchpoint 2: i
(gdb) c
Continuing.
Hardware watchpoint 2: i

Old value = 1108544020
New value = 0
0x0804839d in main () at buggy.c:31
31              for (i=0; i<100; i++) {
(gdb) c
Continuing.
array[i] = one
Hardware watchpoint 2: i

Old value = 0
```


gdb

❑ 단계별 실행 예

```
31              for (i=0; i<100; i++) {
(gdb) n
32                      printf("array[i] = %s\n", array[i]);
(gdb) n
array[i] = two
33                      gtime.hour += gethour(i%2);
(gdb) n
31              for (i=0; i<100; i++) {
```

```
(gdb) s
32                      printf("array[i] = %s\n", array[i]);
(gdb) s
array[i] = three
33                      gtime.hour += gethour(i%2);
(gdb) s
gethour (sw=0) at buggy.c:20
20              if (sw)
(gdb) finish
Run till exit from #0  gethour (sw=0) at buggy.c:20
0x080483df in main () at buggy.c:33
33                      gtime.hour += gethour(i%2);
Value returned is $2 = 4
(gdb)
```

gdb

❑ gdb 명령

- 프로그램 정보 보기: info *keyword*
  - address *name* - *name*의 주소
  - locals - 모든 지역변수
  - variables - 모든 전역변수
  - breakpoints - breakpoint와 watchpoint
  - functions - 모든 함수
  - all-registers - 모든 레지스터
  - frame - 현 스택 프레임 내용 (하나의 함수에 할당된 내역)
  - stack - 스택 역추적 (함수가 스택에 쌓인 순서)

gdb

❑ gdb 명령

- p (print) *name* - 변수 및 레지스터 값 보기
  - *name* 변수의 값 또는 *name* 함수의 주소 출력
    - 포인터 변수의 경우 저장 중인 주소값이 출력됨. 그 주소에 저장된 값을 보려면 간접연산자 * 사용
    - 포인터가 가리키는 구조체 배열 보기 : p *ptr@size
  - 중복된 이름의 변수
    - 기본적으로 현재 실행 중인 함수의 지역변수
    - 특정 변수 지정 - 'file'::var 또는 function::var
  - 출력 형식 설정: p/modifier var
    - modifier는 printf에서 사용되는 것과 거의 같다.
    - t: binary
  - 변수값 설정: p var=value
- display *var* - 변수값의 변화를 자동으로 출력

gdb

```
(gdb) b 32
Breakpoint 4 at 0x80483a5: file buggy.c, line 32.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/juyoon/system/programming/debug/buggy

Breakpoint 4, main () at buggy.c:32
32                      printf("array[i] = %s₩n", array[i]);
(gdb) p array
$7 = {0x80484a4 "one", 0x80484a8 "two", 0x80484ac "three", 0x0}
(gdb) p pt
$8 = (struct time *) 0x80494e0
(gdb) p *pt
$9 = {hour = 1, min = 2, sec = 3}
(gdb) p *pt@4
$10 = {{hour = 1, min = 2, sec = 3}, {hour = 134513828, min = 134513832,
    sec = 134513836}, {hour = 0, min = 1, sec = 1}, {hour = 12,
    min = 134513200, sec = 13}}
(gdb) display i
1: i = 0
(gdb) display gtime.hour
2: gtime.hour = 1
(gdb) s
array[i] = one
33                      gtime.hour += gethour(i%2);
2: gtime.hour = 1
1: i = 0
(gdb) s
gethour (sw=0) at buggy.c:20
20              if (sw)
```


make

- 컴파일 때마다 복잡한 옵션을 일일이 써 주기 싫다!
  - 특히 파일이 많고 구조가 복잡한 경우
  - ➔ 프로젝트 빌드 유틸리티 필요
- make
  - 사용자가 작성한 지시서에 따라 순차적이고 반복적인 작업을 실행하는 명령
    - 컴파일, 문서 생성(DocBook, TeX) 등에 활용된다.
  - 지시서: 일반적으로 Makefile
    - -f 옵션으로 파일 이름을 명시할 수 있다.
    - default는 GNUmakefile, makefile, Makefile

make

❑ Makefile 예제

```
juyoon@ce:~/system/programming/make
[juyoon:make/9]$ ls
합계 16
   4 Makefile     4 print.c     4 test.c     4 test.h
[juyoon:make/10]$ m Makefile
all: t

t: test.o print.o
        gcc -W -Wall -o t test.o print.o -lm

test.o: test.c
        gcc -W -Wall -I../include -c -o test.o test.c

print.o: print.c
        gcc -W -Wall -I../include -c -o print.o print.c

clean:
        rm *.o t
[juyoon:make/11]$ make
gcc -W -Wall -I../include -c -o test.o test.c
gcc -W -Wall -I../include -c -o print.o print.c
gcc -W -Wall -o t test.o print.o -lm
[juyoon:make/12]$ ls
합계 36
   4 Makefile     4 print.o     4 test.c     4 test.o
   4 print.c     12 t*          4 test.h
```

target

dependency

command

make

❑ make의 동작

- 디렉터리 내에서 지시서를 찾는다.
- 지시서 내의 첫 타겟을 찾는다.
- 종속 항목을 디렉터리에서 확인하고, 없으면 명령에 따라 종속 항목을 생성한다.
- 타겟을 지정하면 그 타겟만 생성한다.

```
juyoon@ce:~/system/programming/make
[juyoon:make/14]$ make clean
rm *.o t
[juyoon:make/15]$ ls
합계 16
   4 Makefile     4 print.c     4 test.c     4 test.h
```

make

❑ Makefile 만들기

- 기본 구조

```
CC        = gcc
CFLAGS  = -W -Wall -I../include

target1: dpnd1 dpnd2 #comment
    TAB   command1
          command2

target2: dpnd3 dpnd4 dpnd5
          command3
```

매크로 정의

생성 규칙
명령어

  - 명령어는 무조건 TAB으로 시작해야 한다.
  - 행이 길면 끝에 '\'로 연결, 명령어 분리는 ';'
  - 종속 항목이 없는 타겟도 가능 (예: clean)

make

❑ 매크로

- 사용자 정의 변수에 특정 문자열을 지정
  - 쉘 변수와 유사함

```
NAME = string
```

  - 참조 시에는 $을 붙이고 ( ) 또는 { }로 싼다.

```
target: $(NAME).o
```

  - 중복 정의하면 최후에 지정된 것을 사용
- 내장 매크로 (미리 정의되어 있는 매크로)
  - CC, RM 등은 기본값이 정의되어 있음.
  - CFLAGS, CPPFLAGS, LDFLAGS 등은 이름만 정의됨.

make

❑ 매크로

- 자동매크로

매크로	의 미
$?	현재 타겟보다 최근에 변경된 종속 항목 리스트
$^	현재 타겟의 종속 항목 리스트
$@	현재 타겟의 이름
$<	현재 타겟보다 최근에 변경된 종속 항목 리스트 (확장자 규칙에서만 사용)
$*	현재 타겟보다 최근에 변경된 현재 종속 항목의 이름 (확장자 규칙에서만 사용)

- 내부 정의 매크로와 기본 생성 규칙은 'make -p' 로 볼 수 있다.

make

❑ 확장자 규칙

- 확장자에 따른 기본 동작을 미리 정의해 둠
  - *.o는 같은 이름의 *.c 파일로 만든다 등
- 파일이 많을 때도 간결한 makefile을 만들 수 있다.

```
juyoon@ce:~/system/programming/make
[juyoon:make/124]$ cat Makefile

SRCS     = $(wildcard *.c)        # test.c print.c
OBJECTS  = $(SRCS:.c=.o)          # test.o print.o
CFLAGS   = -W -Wall
LDFLAGS  = -lm

all: t

t: $(OBJECTS)
        $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

clean:
        $(RM) *.o t

[juyoon:make/125]$ make
cc -W -Wall     -c -o print.o print.c
cc -W -Wall     -c -o test.o test.c
cc -W -Wall   -o t print.o test.o -lm
```

make

❑ 여러 디렉터리로 나누어진 경우의 make

- 하나의 Makefile로 make 작동
  - Makefile 내에 직접 디렉터리 지정

```
juyoon@ce:~/system/programming
[juyoon:programming/171]$ cat Makefile3

SRCS    = $(wildcard src/*.c)    # test.c print.c
OBJECTS = $(SRCS:.c=.o)          # test.o print.o
CFLAGS  = -W -Wall -I./include
LDFLAGS = -lm

all: t

t: $(OBJECTS)
        $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

clean:
        $(RM) *.o t
```


make

❑ 여러 디렉터리로 나누어진 경우의 make

- 하나의 Makefile로 make 작동
  - VPATH 매크로 사용
    - 같은 이름의 소스 파일이 있으면 안 된다.
    - recursive clean을 만들기 어렵다.

```
juyoon@ce:~/system/programming
[juyoon:programming/258]$ cat Makefile4

VPATH   = src
OBJECTS = test.o print.o
CFLAGS  = -W -Wall -I./include

t:      $(OBJECTS)
        $(CC) -o $@ $^ -lm
```

❑ 여러 디렉터리로 나누어진 경우의 make

- Recursive Make
  - 각 소스 디렉터리마다 Makefile을 만든다.
  - 최상위 디렉터리의 Makefile에서 각 디렉터리를 방문해 make를 실행하는 구조로 작성한다.

```
DIRS    = src
SRCS    = $(wildcard src/*.c)    # test.c print.c
OBJECTS = $(SRCS:.c=.o)          # test.o print.o
LDFLAGS = -lm
TARGET  = t

all: objs
        $(CC) -o $(TARGET) $(OBJECTS) $(LDFLAGS)

objs:
        @for dir in $(DIRS); do ₩
        make -C $$dir || exit $?; ₩
        done

clean:
        @for dir in $(DIRS); do₩
        make -C $$dir clean; ₩
        done
        rm -rf $(TARGET)
```


❑ 여러 디렉터리로 나누어진 경우의 make

- Recursive Make
  - 각 디렉터리의 Makefile 및 실행 과정

```
juyoon@ce:~/system/programming
[juyoon:programming/307]$ cat src/Makefile

OBJECTS = $(patsubst %.c, %.o, $(wildcard *.c))
CFLAGS  = -W -Wall -I../include

all: $(OBJECTS)
        cp -f $^ ../

clean:
        rm -rf *.o
[juyoon:programming/308]$ make -f Makefile3
make[1]: 들어감 `/home/juyoon/system/programming/src' 디렉토리
cc -W -Wall -I../include    -c -o print.o print.c
cc -W -Wall -I../include    -c -o test.o test.c
cp -f print.o test.o ../
make[1]: 나감 `/home/juyoon/system/programming/src' 디렉토리
cc -o t src/print.o src/test.o                     -lm
```