

삼성 청년 SW 아카데미

리눅스 쉘 프로그래밍

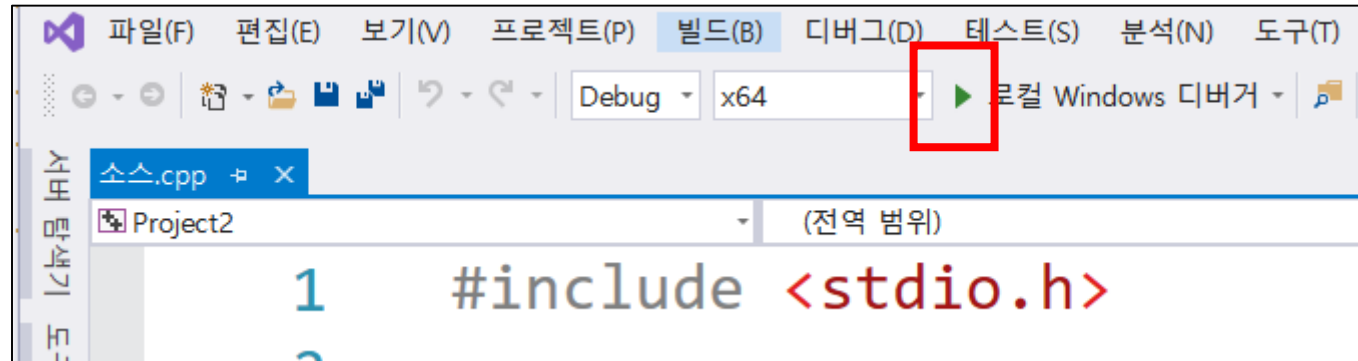
<알림>

본 강의는 삼성 청년 SW아카데미의 컨텐츠로
보안서약서에 의거하여
강의 내용을 어떠한 사유로도 임의로 복사,
촬영, 녹음, 복제, 보관, 전송하거나
허가 받지 않은 저장매체를
이용한 보관, 제3자에게 누설, 공개,
또는 사용하는 등의 행위를 금합니다.

gcc Build Process

빌드란?

- 소스코드에서 실행 가능한 Software로 변환하는 과정 (Process) 또는 결과물



실습을 위한 파일 작성

- green.c 파일
- yellow.c 파일

```
#include <stdio.h>

void yellow();
int main() {

    printf("I'm Green\n");
    yellow();

    return 0;
}
~
~
```

green.c

```
#include <stdio.h>

void yellow() {
    printf("I'm Yellow\n");
}

~
~
```

yellow.c

C언어 빌드 과정 (gcc 기준)

1. Compile & Assemble

- 하나의 소스코드 파일이 0과 1로 구성된 object 파일이 만들어짐

2. Linking

- 만들어진 object 파일들 + Library 들을 모아 하나로 합침

이 두 가지 과정은 **빌드의 대표적인 역할**이다.
더 세부적인 과정은 임베디드 C언어 시간에 다룸

각각의 파일을 Compile & Assemble 하기

- 각각의 c언어 파일을 컴파일(+Assemble) 한다.
- 명령어 수행 (-c 옵션 : Compile and Assemble)
 - `$gcc -c ./green.c`
 - `$gcc -c ./yellow.c`



green.c / yellow.c

각각의 c언어 파일이 존재

Compile &
Assemble



green.o / yellow.o

각각의 파일, Compile 수행

링킹하기

- 만들어진 Object 파일들과 라이브러리 함수들을 하나로 합친다.
 - `$gcc ./green.o ./yellow.o -o ./go`
 - `-o` 옵션 : output 파일 지정



green.o / yellow.o

각각의 Object 파일 (실행 불가)



./go

하나로 합쳐, 하나의 프로그램 생성
(실행 가능)

1. Compile & Assemble 하기



gcc -c



2. Linking 하기

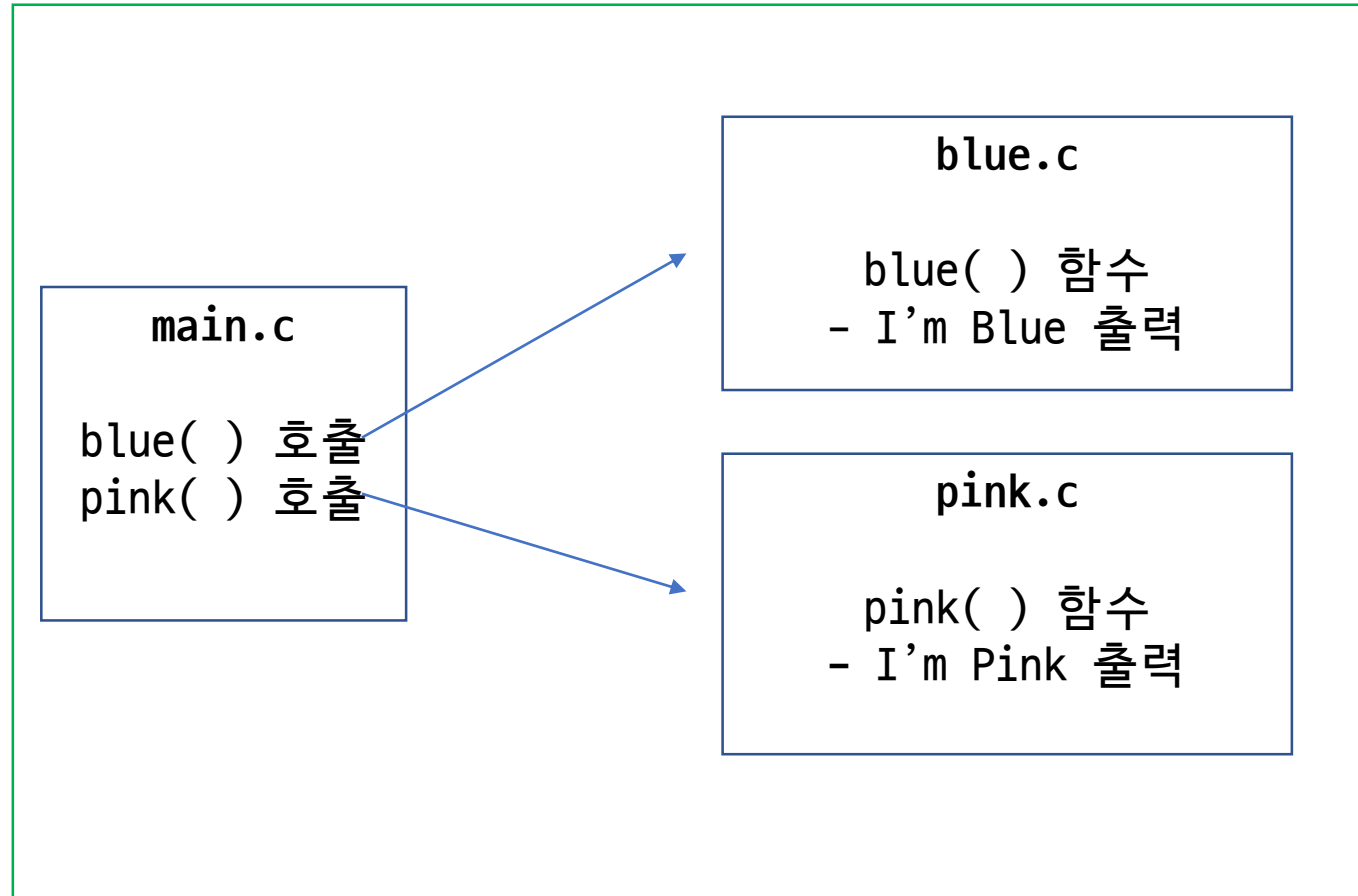
- 실행파일 이름 : **bluepink**



gcc



./hahahoho



물론 gcc가 똑똑해서

아래와 같이 해도 되지만, 학습을 위해 단계를 나누어서 수행한다.

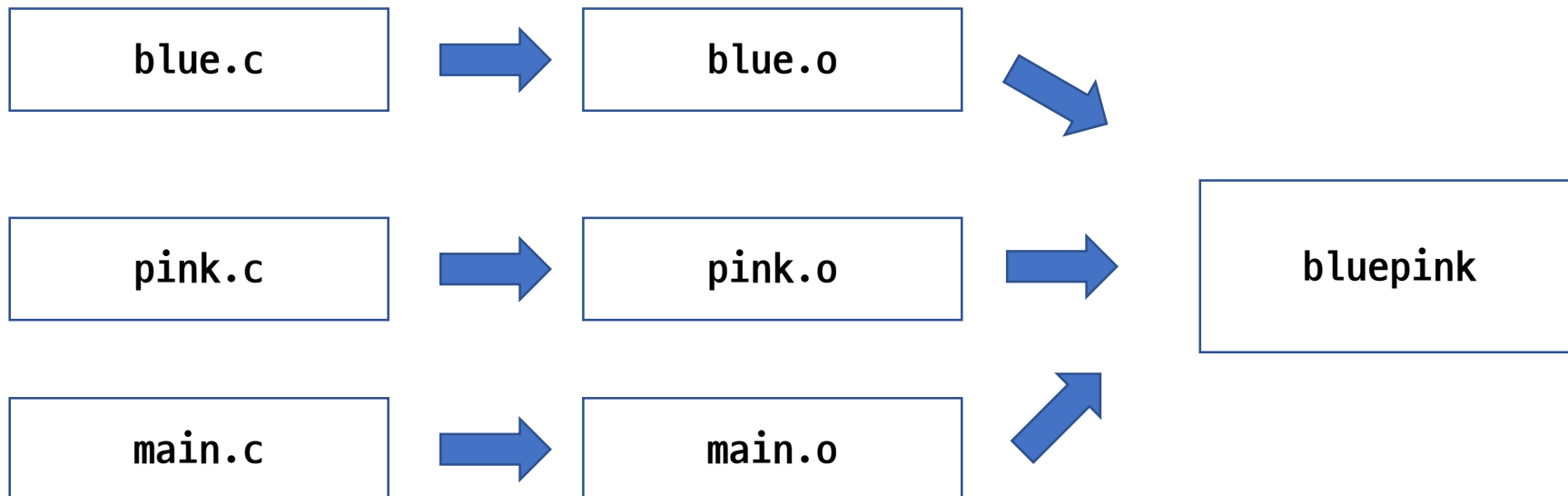
- green.c / yellow.c 를 삭제 후 해야한다.
- a.out : default 값 이름

```
inho@inho:~/work$ gcc ./*.c
inho@inho:~/work$ ./a.out
I'm BLUE
I'm Pink
inho@inho:~/work$
```

빌드 자동화 스크립트

build script 제작하기

- 파일명 : build.sh
 - gcc -c ./blue.c
 - gcc -c ./pink.c
 - gcc -c ./main.c
 - gcc ./main.o ./blue.o ./pink.o -o ./bluepink



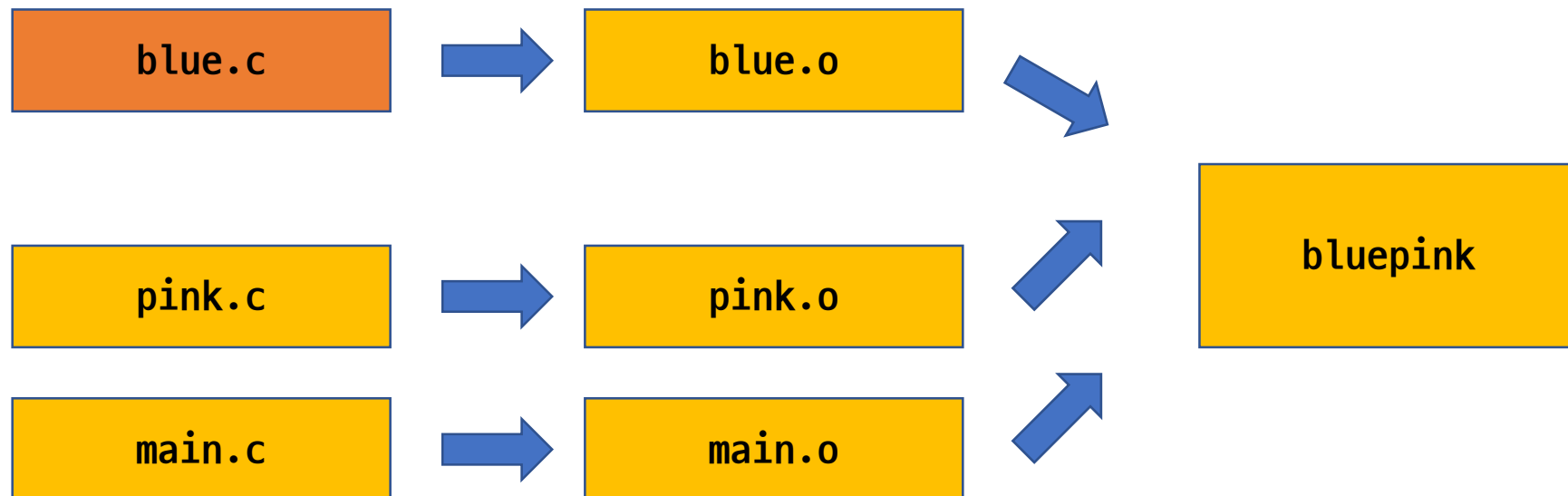
blue.c 파일 하나를 수정하였고, 테스트한다면?

- build.sh 을 다시 수행해주면 됨

→ 모든 파일을 Compile & Assemble 수행함

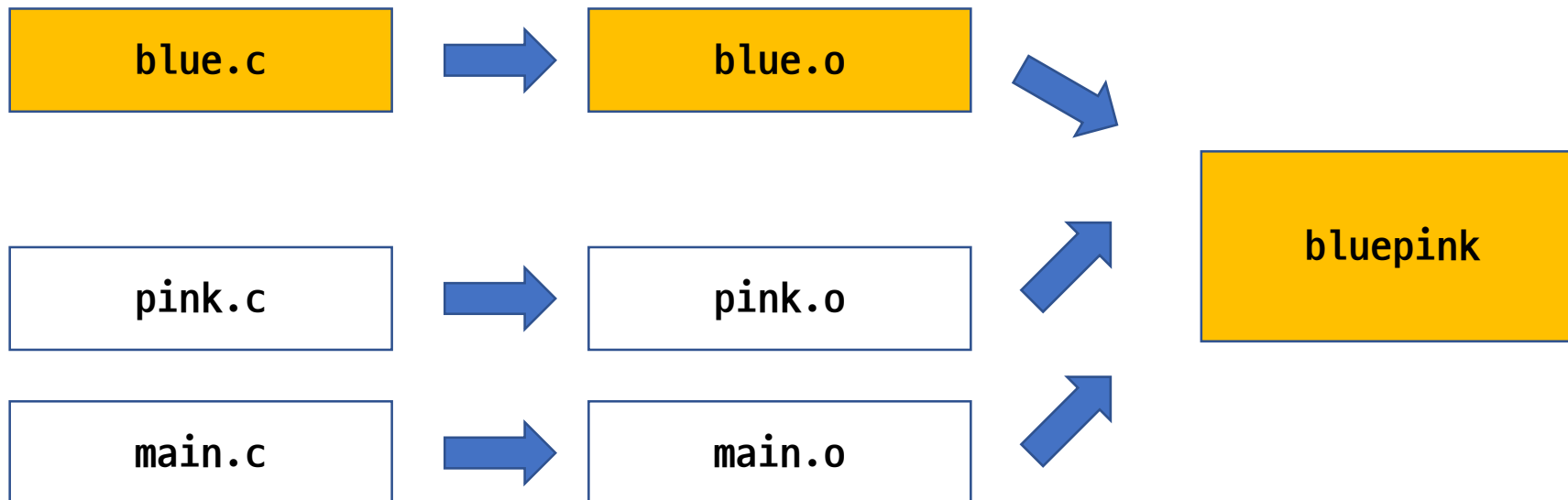
(*기존에 만들었던 모든 Object 파일들이 삭제되고 다시 생성됨)

- rm *.o



blue.c 파일만 변경했다면,
pink.o 파일을 다시 생성할 필요가 없다!

- 그런데 build.sh 파일은 pink.o 파일까지 다시 Compile & Assemble 수행!



Build를 위한 자동 스크립트를 만들 때
Python Script 또는 Bash Shell Script를 사용하지 않는다.

- 필요하지 않는 Compile & Assemble을 수행하여
Build 시간이 오래 걸리게 된다.

make Build System을 쓰면 이 문제를 해결할 수 있다.

build system 체험

Build System 이란?

- Build 할 때 필요한 여러 작업을 도와주는 프로그램들

우리 수업에서 다룰 Build System 종류

1. make
2. cmake

두 가지 Build System에 대해
가볍게 체험해보자!

Build 자동화 스크립트 만드는 방법

1. bash shell script → build 느리다.
2. python script → build 느리다.
3. make build system → 빠르다.

make build system을 체험해보자!

- 설치하기 : `$sudo apt install make -y`

1. “makefile” 이라는 스크립트 파일을 만든다.

- make 문법에 맞추어서 작성해야한다.
- Bash Shell Script 문법과 다르다.

2. 스크립트를 만든 후 스크립트를 실행한다.

- 명령어 : make

기존 object 파일 삭제하기

- `rm -r ./*.o`
- `rm ./bluepink`

makefile 작성

- 파일명 : **Makefile**

반드시
“탭키” 사용

```
inho@inho: ~  
inho@inho: ~ 80x24  
bluepink: blue.o pink.o main.o  
          gcc blue.o pink.o main.o -o bluepink  
blue.o: blue.c  
         gcc -c blue.c  
pink.o: pink.c  
        gcc -c pink.c  
main.o: main.c  
        gcc -c main.c  
~  
~  
~
```

이제 make 을 수행한다.

- make를 해본다. → 빌드 완료

```
inho@inho:~$ make
gcc -c blue.c
gcc -c pink.c
gcc -c main.c
gcc blue.o pink.o main.o -o bluepink
```

- make를 한번 더 해본다. → 빌드 할 필요가 없다고

```
inho@inhopc:~$ make
make: 'bluepink'은(는) 이미 업데이트되었습니다.
inho@inhopc:~$
```

- blue.c 파일만 가볍게 수정 후 make를 해본다.
→ blue.c 만 컴파일 된다

```
inho@inho:~$ make
gcc -c blue.c
gcc blue.o pink.o main.o -o bluepink
inho@inho:~$
```

Make Build System의 장점 두 가지

- Build 자동화
 - 기술된 순서대로 Build 작업을 수행하는 자동화 스크립트 지원
- Build 속도 최적화
 - 불필요한 Compile & Assemble 피하기

QUIZ. Makefile의 필요성?

Confidential

Make가 아래 방법보다 더 좋은 이유가 뭘까?

```
inho@inho:~/work$ gcc ./*.c
inho@inho:~/work$ ./a.out
I'm BLUE
I'm Pink
inho@inho:~/work$
```

CMake

- make 같이 build를 직접적으로 하는 도구가 아니다.
- Makefile을 자동 생성을 할 수 있는 Build System 이다.

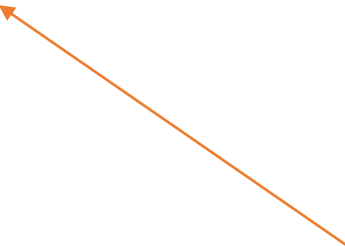
CMake 의 결과물

- Makefile이 만들어진다.

CMake 설치

- `$sudo apt install g++ cmake -y`

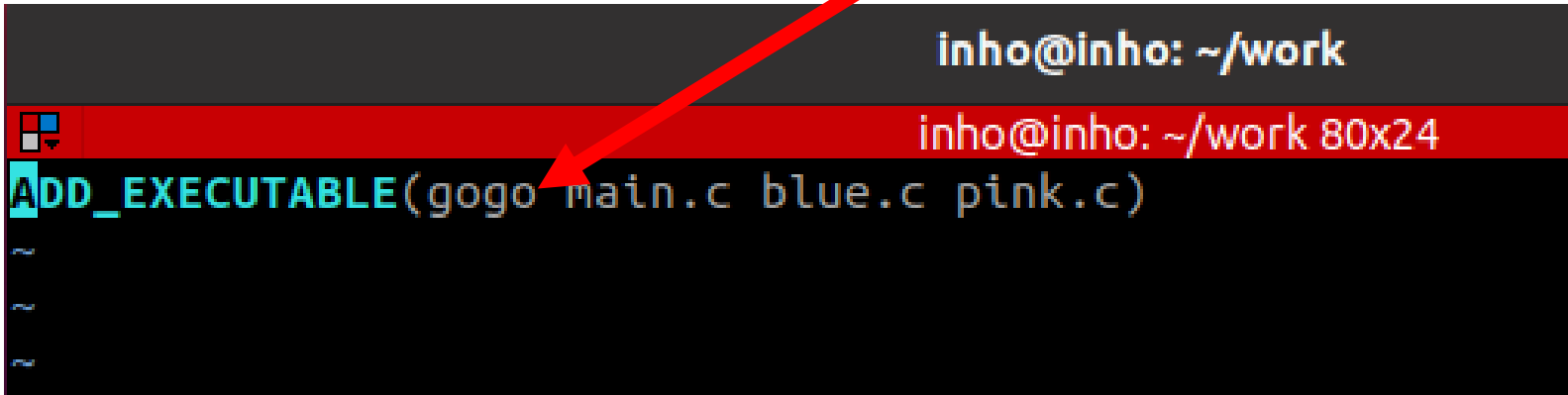
Build System :
빌드 과정에 도움을 주는 툴



CMakeLists.txt 파일 작성

- cmake 하기 위해 CMakeLists.txt 파일이 필요하다.
- 정확히 “CMakeLists.txt” 파일을 하나 생성하자.
 - → 파일명 대소문자 정확하게 하기!!

최종적으로
만들어질 실행파일 이름



```
inho@inho: ~/work
inho@inho: ~/work 80x24
ADD_EXECUTABLE(gogo main.c blue.c pink.c)
~
~
~
```

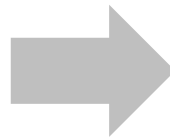
A terminal window with a dark background. The prompt is 'inho@inho: ~/work'. The command 'ADD_EXECUTABLE(gogo main.c blue.c pink.c)' is entered and highlighted in cyan. A red arrow points from the text '최종적으로 만들어질 실행파일 이름' to the word 'gogo' in the command.

cmake 명령어 수행

→ `$cmake .`

- 복잡해 보이는 MakeFile이 자동으로 생성된다.

```
inho@inho:~$ cmake .
```



```
inho@inho:~$ cmake .
CMake Warning (dev) in CMakeLists.txt:
  No project() command is present.  The top-level CMakeLists.txt
  contain a literal, direct call to the project() command.  A
  code such as

    project(ProjectName)

  near the top of the file, but after cmake_minimum_required()
  would be more proper.  This warning can be suppressed by adding
  comment("This file is autogenerated. Do not edit it directly.")
  above the line.
This warning is for project developers.  Use -Wno-dev to suppress it.

-- Configuring done
-- Generating done
-- Build files have been written to: /home/inho
inho@inho:~$
```

위 메시지가 뜨면 성공
Warning 은 있어도 되지만,
Error 가 있으면 안된다.

자동 생성된 Makefile

- \$vi Makefile

```
inho@inho: ~  
inho@inho: ~ 80x24  
# CMAKE generated file: DO NOT EDIT!  
# Generated by "Unix Makefiles" Generator, CMake Version 3.  
  
# Default target executed when no arguments are given to make  
default_target: all  
  
.PHONY : default_target  
  
# Allow only one "make -f Makefile2" at a time, but pass pa  
.NOTPARALLEL:  
  
#=====  
# Special targets provided by cmake.  
  
# Disable implicit rules so canonical targets will work.  
.SUFFIXES:  
  
# Remove some rules from gmake that .SUFFIXES does not remo  
SUFFIXES =  
  
.SUFFIXES: .hpux_make_needs_suffix_list  
"Makefile" 238L, 5742C
```

Build 후 테스트를 해보자

- \$make
- \$./gogo

```
inho@inho:~/work$ make
Scanning dependencies of target gogo
[ 25%] Building C object CMakeFiles/gogo.dir/main.c.o
[ 50%] Building C object CMakeFiles/gogo.dir/blue.c.o
[ 75%] Building C object CMakeFiles/gogo.dir/pink.c.o
[100%] Linking C executable gogo
[100%] Built target gogo
inho@inho:~/work$
inho@inho:~/work$
inho@inho:~/work$ ./gogo
I'm BLUE
I'm Pink
inho@inho:~/work$
```

이번 챕터의 목적

1. **Make** build system 연습
 - Makefile 작성 연습
2. **CMake** build system 연습

make 스크립트 시작

Hello World 출력하기

- Target이 반드시 1 개 이상 존재해야 함
- echo : 화면 출력 shell 명령어

```
1 HI:
2 echo "Hi"
3
4 HELLO:
5 echo "Hello"
```

탭키 필수
(띄어쓰기 불가)

실행방법 두 가지

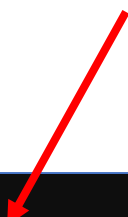
1. `$make HI` : 지정된 Target 수행
2. `$make` : 첫 번째 Target을 수행

```
nojin@nojin-VirtualBox: ~/poo 44x23
1 HI:
2     echo "Hi"
3
4 HELLO :
5     echo "Hello"
6
~
~
~
~

nojin@nojin-VirtualBox: ~/poo 48x24
nojin@nojin-VirtualBox:~/poo$ make
echo "Hi"
Hi
nojin@nojin-VirtualBox:~/poo$ make HI
echo "Hi"
Hi
nojin@nojin-VirtualBox:~/poo$ make HELLO
echo "Hello"
Hello
nojin@nojin-VirtualBox:~/poo$
```

Shell Script 명령어 @

- @ : 수행 할 명령어 입력을 생략하고, 결과만 출력
→ 두 번 출력을 막는다.

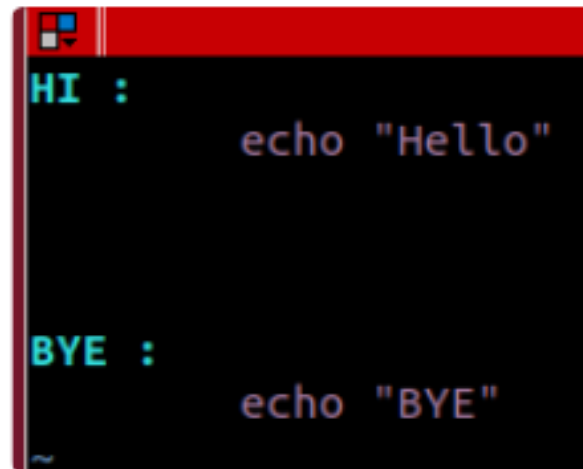


```
1 HI:
2   @echo "Hi"
3
4 HELLO :
5   @echo "Hello"
6
```

```
nojin@nojin-VirtualBox:~/poo$ make
Hi
nojin@nojin-VirtualBox:~/poo$ make HI
Hi
nojin@nojin-VirtualBox:~/poo$ make HELLO
Hello
nojin@nojin-VirtualBox:~/poo$
```

HI 와 BYE Target 만들기

- 각각 수행해보자



```
HI :  
    echo "Hello"  
  
BYE :  
    echo "BYE"
```

의존성 타겟

- 수행 순서를 먼저 예측 후, 실습 진행하기

```
1 HI: one two
2   @echo "Hi"
3
4 HELLO :
5   @echo "Hello"
6
7 one : HELLO
8   @echo "One"
9
10 two :
11   @echo "Two"
12
```

nojin@nojin-VirtualBox:~/poo\$

[실습]

\$make HI

\$make HELLO

\$make one

\$make two

매크로는 글자 그대로 치환된다.

- Make에서는 변수가 아닌 매크로이다.
- \$(MSG1)에 있는 쌍따옴표(“) 까지 글자 그대로 들어간다.
- \$(NONONO) 와 같은 매크로이름은 빈칸으로 출력된다.
- \$make one
- \$make two
- \$make HELLO

```
MSG1 = "One"
MSG2 = "Two"

HI: one two
    @echo "Hi"

HELLO :
    @echo ${asdasd}

one : HELLO
    @echo $(MSG1)

two :
    @echo ${MSG2}
```


MSG3 처럼 매크로는
아무 곳에 넣을 수 있다.

- 가독성을 위해, 최상단에 적어주자.

```
MSG1 = "BBQWORLD"
MSG2 = "ONE"

HI: one two
    @echo $(MSG1);

one:
    @echo $(MSG2);

MSG3 = "HAHA"
two:
    @echo $(MSG3);
```

echo 명령어는 띄워쓰기를 1개만 허용

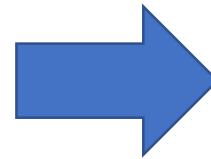
- echo "HI""ABC"
 - HIABC 출력
- echo "HI" "ABC"
 - HI ABC 출력

```
inho@inhopc:~$ echo "HI""ABC"
HIABC
inho@inhopc:~$ echo "HI" "ABC"
HI ABC
inho@inhopc:~$ echo "HI" "ABC" "HIHI"
HI ABC HIHI
inho@inhopc:~$
```

매크로를 사용한 echo 사용

```
KFC="BTS"
HOHO="KFC"

HI:
    echo $(KFC) $(HOHO)
```



```
inho@inhopc:~$ make
echo "BTS" "KFC"
BTS KFC
inho@inhopc:~$
```

[도전] 다음처럼 동작하는 Makefile 만들기

Confidential

make A 입력 시

- #으로 A 문자 만들어 출력하기

```
nojin@nojin-VirtualBox:~/poo/poh$ make
#
##
###
#  #
```

make B 입력 시

- #으로 B 문자 만들어 출력하기

```
nojin@nojin-VirtualBox:~/poo/poh$ make B
##
# #
###
# #
##
```

make All 입력 시

- A, B 문자 모두 출력하기

```
nojin@nojin-VirtualBox:~/poo/poh$ make All
#
##
###
#  #
##
# #
###
# #
##
```

#으로 주석을 나타낸다.

- 구간 설정시
- 명령어 생략시 사용

```
MSG1 = "BBQWORLD"
MSG2 = "ONE"

#=====[gogo]=====

HI: one two
    @echo $(MSG1);

one:
    @echo $(MSG2);

#=====[haha]=====
MSG3 = "HAHA"
two:
    @echo $(MSG3);
```

+= 기호로, 기존 매크로 내용에 추가된다.

- += 할 때 마다, 띄어쓰기 한 칸이 자동으로 추가 됨

```
NAME = "OH"  
NAME += "GOOD"  
NAME += "KFC"  
  
who :  
    @echo $(NAME)
```

두 가지 대입 연산자

- Simple Equ (:=)
 - Script 순서대로 현재 기준에서 값을 넣는다.
 - \$(SIMPLE) 출력 결과 : OH
- Recursive Equ (=)
 - 최종 변수 결과를 집어 넣는다.
 - \$(RECUL) 출력 결과 : OH GOOD KFC

```
NAME = "OH"

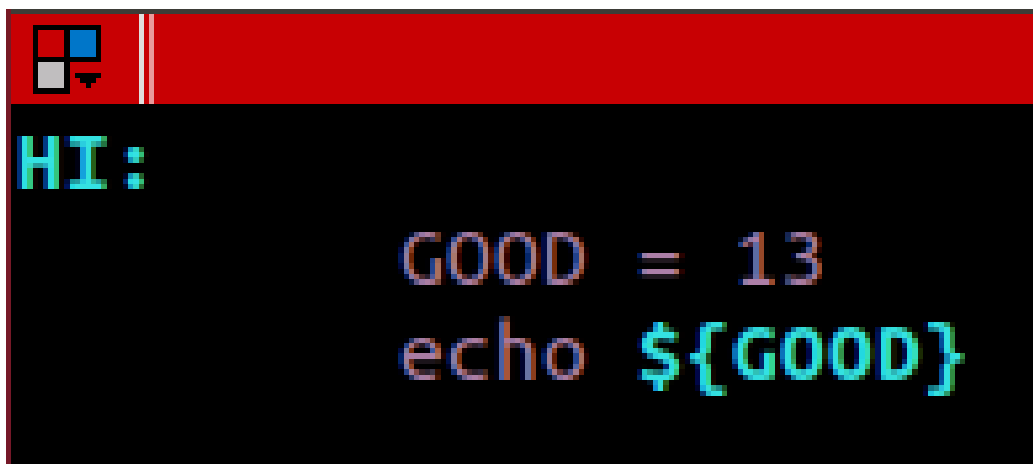
SIMPLE := $(NAME)
RECUL = $(NAME)

NAME += "GOOD"
NAME += "KFC"

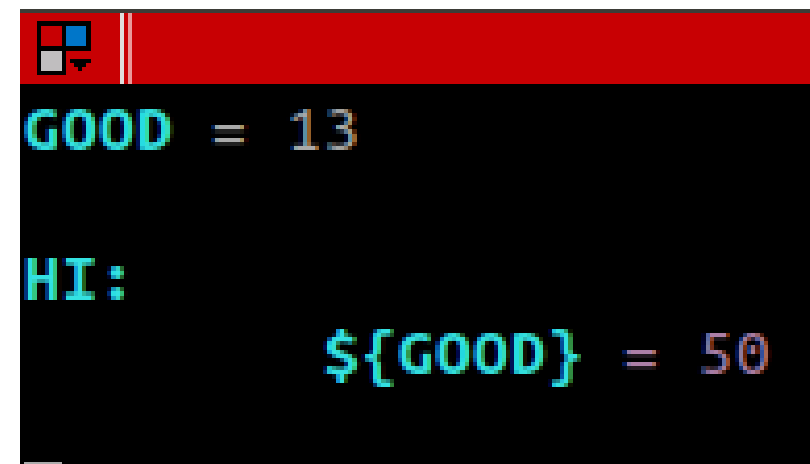
who:
    @echo $(SIMPLE)
    @echo $(RECUL)
```

Shell 명령어와 Make Script 를 구분하자

- 틀린 코드를 찾아 수정



```
HI:
    GOOD = 13
    echo ${GOOD}
```



```
GOOD = 13
HI:
    ${GOOD} = 50
```

echo 는 문자 그대로 출력된다.

- `echo 13 + 55`
 - 출력결과 : 13 + 55
- `echo "BTS" ABC BB`
 - 출력결과 : BTS ABC BB
- `echo 'ABC'` `bbq`
 - 출력결과 : ABC bbq

\$make HI / \$make HE 를 수행했을 때 출력 결과 예측하기

```
GOOD = "BTS"

HI :
    @echo $(GOOD)

GOOD += "KFC"
```

```
GOOD = "BTS"

HI :
    @echo $(GOOD)

GOOD += "KFC"
GOOD = "BBQ"
```

```
GOOD += 13

HI :
    @echo $(GOOD)

GOOD += "KFC"
```

```
HE :
    @echo $(GOOD)
```

```
GOOD += 13

HI :
    @echo $(GOOD)
```

```
GOOD += "KFC"
GOOD := 20
```

```
HE :
    @echo $(GOOD)
```

두 가지 실행결과를 예측해보자.

```
GOOD += "BTS"
GOOD := "ABC"
GOOD = "KFC"

HI:
    echo ${GOOD}

GOOD += "T"

HE:
    echo ${GOOD}
```

```
GOOD += "BTS"
GOOD += "ABC"
BTS = ${GOOD}
HOT := ${GOOD}

HI:
    echo ${BTS}

GOOD = "T"

HE:
    echo ${HOT}
```

자동화 스크립트 제작

shell 명령어를 모아, 자동화 스크립트 제작 가능

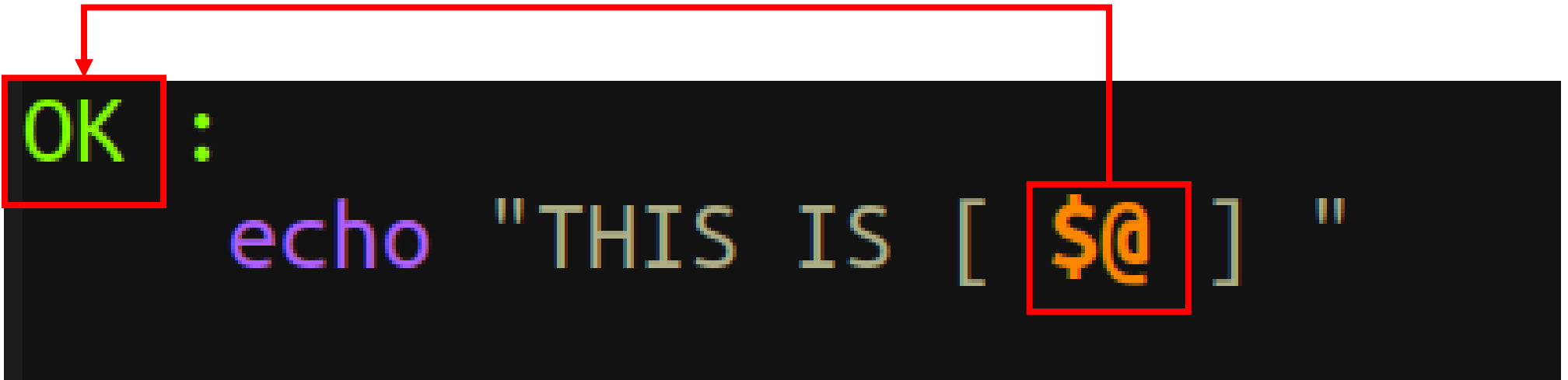
- 순차적으로 명령을 수행한다.
- \$make A
- \$make B

```
A :  
    mkdir hi  
    cd hi  
    ls
```

```
B :  
    cd ..  
    rmdir hi  
    ls
```

`$@` = Target을 나타내는 변수

- `$make OK`



```
OK: echo "THIS IS [ $@ ] "
```

make write 입력시

- “[write] 명령어 수행 중..” 출력하기
- dd로 파일 쓰기 : `dd if=/dev/zero of=./dummy.dat bs=10M count=1`
- “[write] 명령어 수행 완료” 출력하기

make read 입력시

- “[read] 명령어 수행 중..” 출력하기
- dd로 파일 읽기 : `dd if=./dummy.dat of=/dev/null bs=1024`
- “[read] 명령어 수행 완료” 출력하기

make clean 입력시

- dummy.dat 파일 삭제
- “[clean] 완료” 출력하기

\$@ 를 사용하여 make 생성

GCC MakeFile

다음 소스코드를 준비한다.

go.c : 메인 함수

```
1 #include <stdio.h>
2 #include "hi.h"
3
4 int main()
5 {
6     hello(1, 2, 3);
7
8     return 0;
9 }
```

hi.h

```
1 #include <stdio.h>
2
3 void hello(int a, int b, int c);
~
```

hi.c : 출력 함수

```
1 #include "hi.h"
2
3 void hello(int a, int b, int c)
4 {
5     printf("START\n");
6     printf("a = %d\n", a);
7     printf("b = %d\n", b);
8     printf("c = %d\n", c);
9     printf("END\n");
10 }
```


타겟 : 의존성

- 최종적으로 go 파일을 생성하게끔 하는 Makefile
- \$make
- \$./go

```
1 go : go.o hi.o
2     gcc -o go ./go.o ./hi.o
3
4 go.o : go.c
5     gcc -c ./go.c
6
7 hi.o : hi.c
8     gcc -c ./hi.c
```



```
inho@com:~/jason$ ./go
START
a = 1
b = 2
c = 3
END
inho@com:~/jason$
```

컴파일러를 변수로 변경하기

- 실제 임베디드 환경에서는
빌드해야하는 툴이 gcc가 아니라 다른 프로그램을 선택해야 할 때가 있다.
(ARM ToolChain)

```
1 CC = gcc
2
3 go : go.o hi.o
4     $(CC) -o go ./go.o ./hi.o
5
6 go.o : go.c
7     $(CC) -c ./go.c
8
9 hi.o : hi.c
10    $(CC) -c ./hi.c
```

내부 매크로 사용

- \$@ : Target을 나타냄
- \$^ : 의존성 파일들을 나타냄

```
1 CC = gcc
2
3 go : go.o hi.o
4     $(CC) -o $@ $^
5
6 go.o : go.c
7     $(CC) -c $^
8
9 hi.o : hi.c
10    $(CC) -c $^
```

실행파일 매크로 추가

- RESULT 추가
- clean 타겟 추가 (반복 실험을 편하도록 clean

```
1 CC = gcc
2 RESULT = go
3
4 go : go.o hi.o
5     $(CC) -o $@ $^
6
7 go.o : go.c
8     $(CC) -c $^
9
10 hi.o : hi.c
11     $(CC) -c $^
12
13 clean :
14     rm ./*.o ./$(RESULT)
```

단계별 makefile 제작

make파일을 한줄한줄 완성해 나가자

- 비어 있는 디렉토리에서 시작한다.
- 제로베이스에서 시작한다.



a.h



b.h



c.h



test1.

c



test2.c



test3.c

a.h / b.h / c.h 파일 작성하기

Confidential



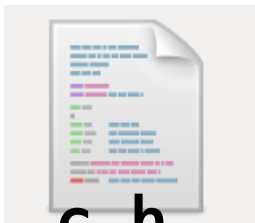
a.h

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



b.h

```
#define N1 11
```

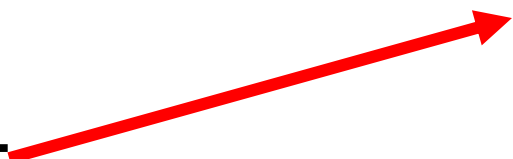


c.h

```
#define N2 22
```

#include 역할

해당 파일을
그대로 복사 붙여넣
기



```
#include "a.h"
#include "b.h"

void func1();
void func2();

int main() {

    int n = N1;
    printf("TEST 1 : %d\n", n);

    func1();
    func2();

    return 0;
}
```


Build 과정 中
전처리작업 이
후
변화

```
info@inhopc: ~  
#include "a.h"  
#include "b.h"  
  
void func1();  
void func2();  
  
int main() {  
    int n = N1;  
    printf("TEST 1 : %d\n", n);  
  
    func1();  
    func2();  
  
    return 0;  
}
```

변경됨

```
info@inhopc: ~  
#include <stdio.h>  
#define N1 11  
  
void func1();  
void func2();  
  
int main() {  
    int n = N1;  
    printf("TEST 1 : %d\n", n);  
  
    func1();  
    func2();  
  
    return 0;  
}
```

오타자 유의

```
#include "a.h"  
#include "c.h"
```

```
void func1() {
```

```
    int n = N2;
```

```
    printf("TEST 2 : %d\n", n);
```

```
}
```

오탈자 유의

```
#include "a.h"
#include "b.h"
#include "c.h"

void func2() {

    int n = N1 + N2;
    printf("TEST 3 : %d\n", n);
}
```

1단계

- Makefile 만들기 →
- 빌드 테스트 해보기
- \$make
- \$./result

```
result : test1.o test2.o test3.o
        gcc -o result test1.o test2.o test3.o

test1.o : test1.c a.h b.h
        gcc -c test1.c

test2.o : test2.c a.h c.h
        gcc -c test2.c

test3.o : test3.c a.h b.h c.h
        gcc -c test3.c
```

2단계

- clean 추가
 - 빌드 과정에 생기는 파일들 제거를
- 테스트 방법
 - \$make clean

```
result : test1.o test2.o test3.o
        gcc -o result test1.o test2.o test3.o

test1.o : test1.c a.h b.h
        gcc -c test1.c

test2.o : test2.c a.h c.h
        gcc -c test2.c

test3.o : test3.c a.h b.h c.h
        gcc -c test3.c

clean :
        rm test1.o test2.o test3.o result
```

3단계

- 매크로 추가

```
CC = gcc
OBJS = test1.o test2.o test3.o

result : $(OBJS)
    $(CC) -o result $(OBJS)

test1.o : test1.c a.h b.h
    $(CC) -c test1.c

test2.o : test2.c a.h c.h
    $(CC) -c test2.c

test3.o : test3.c a.h b.h c.h
    $(CC) -c test3.c

clean :
    rm $(OBJS) result
```

테스트 1

- `$make clean`
- `$make`
- `$make`

테스트 2

- `$touch test1.c`
 - 최신 날짜로 변경한다. → make에서는 파일이 변경되었다고 인식한다.
- `$make`

테스트 3

- `$touch c.h`
- `$make`

4단계

- 내부 매크로 사용하기 1
 - `$@` : Target을 나타냄
 - `$$` : 의존성 파일들을 나타냄
 - `$<` : 의존 파일 中 첫 번째 파일을 나타냄
- 완성 후 테스트 할 것.
- `$make clean`
- `$make`
- `./result`

```
CC = gcc
OBJS = test1.o test2.o test3.o

result : $(OBJS)
    $(CC) -o $@ $$

test1.o : test1.c a.h b.h
    $(CC) -c $<

test2.o : test2.c a.h c.h
    $(CC) -c $<

test3.o : test3.c a.h b.h c.h
    $(CC) -c $<

clean :
    rm $(OBJS) result
```


5단계

- 컴파일러 지정 \$(CC)
- 컴파일 옵션 지정 \$(CFLAGS)
 - -g : 디버깅 (Trace) 가능하도록 설정

```
CC = gcc
CFLAGS = -g
OBJS = test1.o test2.o test3.o

result : $(OBJS)
    $(CC) -o $@ $^

test1.o : test1.c a.h b.h
    $(CC) $(CFLAGS) -c $<

test2.o : test2.c a.h c.h
    $(CC) $(CFLAGS) -c $<

test3.o : test3.c a.h b.h c.h
    $(CC) $(CFLAGS) -c $<

clean :
    rm $(OBJS) result
```

옵션 추가하기

- -Wall : Warning 이 뜨면 Error처럼 멈추도록 함
- -O2 : 최적화 2단계 옵션

테스트하기

- \$make clean
- \$make
- \$./result

```
CC = gcc
CFLAGS = -g -Wall -O2
OBJS = test1.o test2.o test3.o
```

wildcard, 확장자 치환 사용

- 출력 테스트용
temp 태그를 하나 더 만들기
- 확장자 치환 테스트
 - \$make temp

아래 코드 있음



```
CC = gcc
CFLAGS = -g -Wall -O2
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
```

```
result : $(OBJS)
$(CC) -o $@ $^
```

```
temp:
    @echo $(SRCS)
    @echo $(OBJS)
```

테스트를 위한 태그 삭제

- temp 태그 제거

```
CC = gcc
CFLAGS = -g -Wall -O2
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)

result : $(OBJS)
        $(CC) -o $@ $^

test1.o : test1.c a.h b.h
        $(CC) $(CFLAGS) -c $<

test2.o : test2.c a.h c.h
        $(CC) $(CFLAGS) -c $<

test3.o : test3.c a.h b.h c.h
        $(CC) $(CFLAGS) -c $<

clean :
        rm $(OBJS) result
```

Makefile 디렉토리에 있는
모든 .c 파일은
컴파일 대상이 된다.
→ 이것은 편리

각각의 c 파일이 사용
하는
header 파일들을
하나씩 입력해준다.
→ 이것은 불필

```
CC = gcc
CFLAGS = -g -Wall -O2
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)

result : $(OBJS)
    $(CC) -o $@ $^
```

```
test1.o : test1.c a.h b.h
    $(CC) $(CFLAGS) -c $<

test2.o : test2.c a.h c.h
    $(CC) $(CFLAGS) -c $<

test3.o : test3.c a.h b.h c.h
    $(CC) $(CFLAGS) -c $<
```

```
clean :
    rm $(OBJS) result
```

이 부분을 자동으로 생성 가능한 방법

- 두 가지 작업을 해주면 된다.
 - 8 단계 : makedepend 유틸리티를 사용
 - 9 단계 : SUFFIXES 추가 해주기

```
CC = gcc
CFLAGS = -g -Wall -O2
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)

result : $(OBJS)
$(CC) -o $@ $^

test1.o : test1.c a.h b.h
$(CC) $(CFLAGS) -c $<

test2.o : test2.c a.h c.h
$(CC) $(CFLAGS) -c $<

test3.o : test3.c a.h b.h c.h
$(CC) $(CFLAGS) -c $<

clean :
rm $(OBJS) result
```

makedepend

- 입력한 c 파일을 분석하여
의존성 헤더파일을 등록해주는 make 도우미 유틸리티
- 설치해보자.
 - `$sudo apt install xutils-dev`

테스트 해보기

- `$makedepend test1.c test2.c test3.c -Y`
 - `-Y` 옵션 : 분석할 파일 경로를, 현재 디렉토리로 한다.

```
nojin@nojin-VirtualBox:~/poo$ makedepend test1.c test2.c test3.c -Y
makedepend: warning: test1.c (reading a.h, line 1): cannot find include
file "stdio.h"
```

- `$vi Makefile`

```
clean :
    rm $(OBJS) result
```

```
# DO NOT DELETE
```

```
test1.o: a.h b.h
test2.o: a.h c.h
test3.o: a.h b.h c.h
```


dep 태그 추가하기

- makedepend 명령어가 실행 될 수 있도록 dep 태그 추가

```
test3.o : test3.c a.h b.h c.h  
$(CC) $(CFLAGS) -c $<
```

```
clean :  
rm $(OBJS) result
```

```
dep :  
makedepend $(SRCS) -Y
```

SUFFIXES

• .SUFFIXES : .c .o

- .c 파일을 사용하여 .o 파일을 만들어야 할 때 자동으로 호출되는 태그를 지정한다.

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

```
result : $(OBJS)
```

```
@echo "==<FINISH>=="
```

```
$(CC) -o $@ $^
```

```
.c .o :
```

```
@echo "==[READY]=="
```

```
$(CC) $(CFLAGS) -c $<
```

```
clean :
```

```
rm $(OBJS) result
```

테스트 1

- `$make clean`
- `$make`
- `$make`

테스트 2

- `$touch test1.c`
 - 최신 날짜로 변경한다. → make에서는 파일이 변경되었다고 인식한다.
- `$make`

테스트 3

- `$touch c.h`
- `$make`

Default SUFFIXES 값

- 사실 .c.o 관련된 SUFFIXES 는 Makefile 에 기본값 설정이 되어있다.
- 따라서, 해당 코드를 지우더라도 빌드가 잘 된다.

테스트 해보자.

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

```
result : $(OBJS)  
    @echo "==<FINISH>=="  
    $(CC) -o $@ $^
```

```
#.c .o :  
#    @echo "==[READY]=="  
#    $(CC) $(CFLAGS) -c $<
```

```
clean :  
    rm $(OBJS) result
```

결과 파일명을 지정해주는
매크로를 추가해주자.

```
CC = gcc
CFLAGS = -g -Wall -O2
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
TARGET = result

all : $(OBJS)
    @echo "==<FINISH>=="
    $(CC) -o $(TARGET) $^

clean :
    rm $(OBJS) $(TARGET)

dep :
    makedepend $(SRCS) -Y
```

사용방법

1. c 파일이 있는 곳에 해당 Makefile 놓아둔다.
2. \$make clean을 해준다.
3. \$make dep 해준다.
4. \$make 를 한다.
5. \$./result

만들어지는 결과파일

- result

```
CC = gcc
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
CFLAGS = -g -Wall -O2
SUFFIXES = .c.o
TARGET = result

all: $(OBJS)
    @echo "==<FINISH>=="
    $(CC) -o $(TARGET) $^

.c.o:
    @echo "==[READY]== "
    $(CC) $(CFLAGS) -c $<

clean:
    rm -rf $(OBJS) result

dep:
    makedepend $(SRCS) -Y
```

수업의 목표

먼저, 파일을 다루는 방법을 배운다.

- 파일 정보 확인 방법, 파일 검색하기
- 바로가기 파일 만들기 (링크 파일),
- 파일 압축 하기

그리고 외부 프로그램들을
내 리눅스 시스템에 설치하는 방법을 배운다.

- apt를 사용하지 않고, 설치하는 방법을 학습

파일 관리시 자주 사용되는 Shell 명령어

목적Goal

사용 빈도가 높고 중요한 리눅스 셸 명령어를 배우자.
기존에 배운 명령어와 조합해서 연습해보자.

1. cat
2. grep
3. find
4. history

cat 명령어

- 파일 내용을 출력한다.
- `$cat /proc/cpuinfo`
 - `proc` 디렉토리에는 시스템 정보들이 모아져있다.
- 내용을 파일로 저장하는 방법
 - 명령어 뒤에 “> `bts.txt`” 를 붙인다.
 - `$cat /proc/cpuinfo > bts.txt`

```
inho@inho-VirtualBox:~$  
inho@inho-VirtualBox:~$ cat /proc/cpuinfo  
processor       : 0  
vendor_id      : GenuineIntel  
cpu family     : 6  
model          : 126  
model name     : Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz  
stepping       : 5  
cpu MHz        : 1497.591  
cache size     : 8192 KB  
physical id    : 0  
siblings       : 4  
core id        : 0  
cpu cores      : 4  
apicid         : 0  
initial apicid : 0  
fpu            : yes  
fpu_exception  : yes  
cpuid level    : 22  
wp             : yes
```

[도전] 시스템 정보 저장하기

다음 내용을 처리하자.

1. /proc/cpuinfo 내용을 cpu.txt로 저장
 - 저장된 파일에서 “processor” 를 검색하여, 몇 개가 검색되는지 확인한다.
2. /proc/meminfo 내용을 mem.txt로 저장
 - “Percpu” 가 몇 Kb 인지 검색한다.
 - Percpu 라인을 8번 복사해서 붙여넣기를 한다.
 - 모든 kB를 “kG” 으로 변경한다.
3. cpu.txt 와 mem.txt 의 이름을 cpuinfo, meminfo 로 변경한다.
(다음 미션을 위해 삭제하지 않는다.)

```
inho@inho-VirtualBox: ~  
29 Bounce: 0 kB  
30 WritebackTmp: 0 kB  
31 CommitLimit: 4112012 kB  
32 Committed_AS: 3774848 kB  
33 VmallocTotal: 34359738367 kB  
34 VmallocUsed: 33596 kB  
35 VmallocChunk: 0 kB  
36 Percpu: 2768 kB  
37 Percpu: 2768 kB  
38 Percpu: 2768 kB  
39 Percpu: 2768 kB  
40 Percpu: 2768 kB  
41 Percpu: 2768 kB  
42 Percpu: 2768 kB  
43 Percpu: 2768 kB  
44 Percpu: 2768 kB  
45 HardwareCorrupted: 0 kB  
46 AnonHugePages: 0 kB  
47 ShmemHugePages: 0 kB  
48 ShmemPmdMapped: 0 kB  
49 FileHugePages: 0 kB  
50 FilePmdMapped: 0 kB  
./mem.txt [+]  
1 line less; before #1 1 초 전
```

find

- 파일을 찾는 명령어
- find [경로] -name “파일명”
 - 전 영역에서 kfc.txt 라는 파일 찾기 : `$find / -name “kfc.txt”`
 - 단어에 config가 포함된 모든 파일, 디렉토리 찾기 : `$find ./ -name “*config*”`
- 파일 or 디렉토리만 찾기 옵션
 - 파일만 찾기 : `find [경로] -name “파일명” -type f`
 - 디렉토리만 찾기 : `find [경로] -name “파일명” -type d`

```
inho@inho-VirtualBox:/bin$ sudo find . -name "z*"
./zip
./zipsplit
./zmore
./zdiff
./zless
./zcat
```

[도전] 파일을 찾아보자

다음 명령어 순서대로 처리해보자.

1. /proc 디렉토리에 info 로 시작하는 파일들 찾아보기 (디렉토리 제외)

- 관리자 권한으로 검색할 것
- proc 디렉토리 : 시스템 정보들을 읽을 수 있는 특수한 디렉토리

2. /bin 디렉토리에 z로 시작하는 파일들을 모두 찾아

- 경로를 적을때 뒤에 / 를 붙여주어야 한다. /bin/
 - /bin 은 링크이기 때문에 / 를 뒤에 붙여야한다.
 - [TIP] find 에서 경로 뒤에 항상 '/' 를 붙여주는 것을 잊지말자!

```
inho@inho-VirtualBox:~$ sudo find /bin -name "info*"
inho@inho-VirtualBox:~$ sudo find /usr/bin -name "info*"
/usr/bin/infocmp
/usr/bin/infotocap
/usr/bin/info
/usr/bin/infobrowser
inho@inho-VirtualBox:~$
```

힌트. find 명령어 예시

grep

- 문자열 검색
- 내용에 문자열을 검색하고 싶을때 사용한다.
 - 정규표현식 사용 가능 (정규표현식은 차후 학습 예정)
- meminfo 파일 내 “kB” 문자열 검색
 - `$grep kB ./meminfo`
 - kB 문자열이 포함된 한 줄 라인이 출력된다.
- 현재 디렉토리에 있는 모든 파일을 대상으로, “ache” 가 들어있는 라인 모두 출력
 - `$grep ache ./ * -r`
 - r 옵션 : 내부 디렉토리에 있는 파일들도 모두 찾는다.
- 다른 명령어 출력결과에 문자열 검색하기 (bar를 파이프라고 부름)
 - `$ls -al | grep .v`
 - ls -al 출력 결과를 오른쪽으로 넘겨서 처리하는 원리
예시. `$cat abc.txt | grep bbq`



```
inho@inho-VirtualBox:~$ ls -al | grep .v
-rw-r----- 1 inho inho      5  3월  3 19:53 .vboxclient-clipboard.pid
-rw-r----- 1 inho inho      5  3월  3 19:53 .vboxclient-display-svga-x11.pid
-rw-r----- 1 inho inho      5  3월  3 19:53 .vboxclient-draganddrop.pid
-rw-r----- 1 inho inho      5  3월  3 19:53 .vboxclient-seamless.pid
-rw----- 1 inho inho    9409  3월  3 20:16 .viminfo
-rw-rw-r--  1 inho inho     50  3월  2 16:28 .vimrc
drwxrwxr-x  3 inho inho   4096  3월  1 14:33 .vscode-insiders
inho@inho-VirtualBox:~$
```

[도전] 내용 검색하기

다음 명령을 30초 컷으로 수행해보자

1. /proc/meminfo 파일에 “File” 문자열 검색하는 두 가지 방법 모두 수행
2. Home Directory 에서 ls -al 수행 결과물에 “4096” 이 포함된 라인만 출력하기
3. “zip” 프로그램이 시스템 어디에 있는지 찾기
 - 관리자 권한으로 전체 검색한다.
4. “.vimrc” 파일이 시스템 어디에 있는지 찾고, 결과를 ~/bts.txt에 저장한다.
 - 관리자 권한으로 전체 검색한다.
5. “linux” 디렉토리가 시스템 어디에 있는지 찾고, 경로에 “include” 가 있어야 한다.
 - 관리자 권한, 디렉토리만 검색한다.
 - find와 grep을 동시에 이용

이력들이 출력된다

- !번호 : 특정번호 명령어가 수행됨
- 예시
 - !720
 - !723

```
inho@
701 cmake -v
702 cmake -ver
703 cmake -version
704 vi CMakeLists.txt
705 cmake .
706 vi Makefile
707 vi CMakeLists.txt
708 cmake .
709 vi Makefile
710 make
711 ./result
712 touch c.h
713 make
714 touch
715 touch c.h
716 make
717 ./result
718 make help
719 make depend
720 vi Makefile
721 make result
722 vi CMakeLists.txt
723 history
inho@inhopc:~$ S
```

심볼릭 링크

챕터목적

- 링크의 종류 보다는, 심볼릭 링크(바로가기)를 간단히 만들어보고, 활용하는 방법에 포커스를 맞춘다.
 - 아이노드 / 하드링크 내용 제외

심볼릭 링크를 만들어보자.

1. ls 바이너리의 위치를 찾자 (which 명령어)
 - /usr/bin/ls
2. ln 명령어 (링크)
 - -s 옵션 : 심볼릭
3. bts 심볼릭링크(바로가기) 를 실행하면 ls가 실행되도록 하자.
 - ln -s /usr/bin/ls bts
4. bts를 실행해보자.
5. ls -al ./bts 확인해보기

```
inho@inho-VirtualBox:~$ ls -al ./bts
lrwxrwxrwx 1 inho inho 11 3월 6 01:10 ./bts -> /usr/bin/ls
```

심볼릭 링크로 가능과 한계 여부 직접 실험을 해보고, 정답을 적어보자.

1. show.txt 파일을 만들고, 내용을 적었다.
kfckfc 심볼릭링크를 만들었다. (위 파일과 연결)
vi kfckfc 라고 했을 때, show.txt 파일이 열리고, 편집이 가능할까? (O 또는 X)
2. 디렉토리를 심볼릭링크로 만들었다.
“cd 심볼릭링크명” 입력했을 때, 해당 디렉토리로 이동할까? (O, X)
3. date 를 심볼릭링크로 만들었다.
심볼릭링크를 한번 더 심볼릭링크로 만들었다.
이렇게 했을 때도, 실행이 될까? (O, X)

셸에서 다음 명령어를 수행해보자.

1. 현재 사용자명 출력
 - 명령어 : `users`
2. 현재 Host명 출력
 - 명령어 : `hostname`
3. 부팅 후 시간 출력하기
 - 명령어 : `uptime -p`
4. 로그인 가능한 user 목록들 출력하기
 - 명령어 : `cat etc/passwd | grep bash`

위 순서대로 동작하는 C언어 프로그램을 작성해보자.

- 프로그램 이름 : `now`

now 프로그램 심볼릭링크 만들기

- now라는 심볼릭링크를 /usr/bin/ 내부에 만든다.
 - now 위치를 절대경로로 입력해야 한다.
 - 예시) `ln -s ~/work/now /usr/bin/now`
- /usr/bin 에 넣으면, 어디서든 프로그램 실행이 가능하다.

아무 디렉토리로 이동한 후, now 라고 실행해보자.

- `cd /home`
- `now`
- `cd /proc`
- `now`

./now 와 now 의 차이를 이해하자.

```
inho@inho-VirtualBox:/proc$ now
inho
inho-VirtualBox
up 1 day, 43 minutes
root:x:0:0:root:/root:/bin/bash
inho:x:1000:1000:inho,,,:/home/inho:/bin/bash
carrot:x:1001:1003:,,,:/home/carrot:/bin/bash
jplace:x:1002:1004:,,,:/home/jplace:/bin/bash
onnew:x:1003:1005:,,,:/home/onnew:/bin/bash
tigerstar:x:1004:1006:,,,:/home/tigerstar:/bin/bash
inho@inho-VirtualBox:/proc$
```

파일 정보 확인 명령어

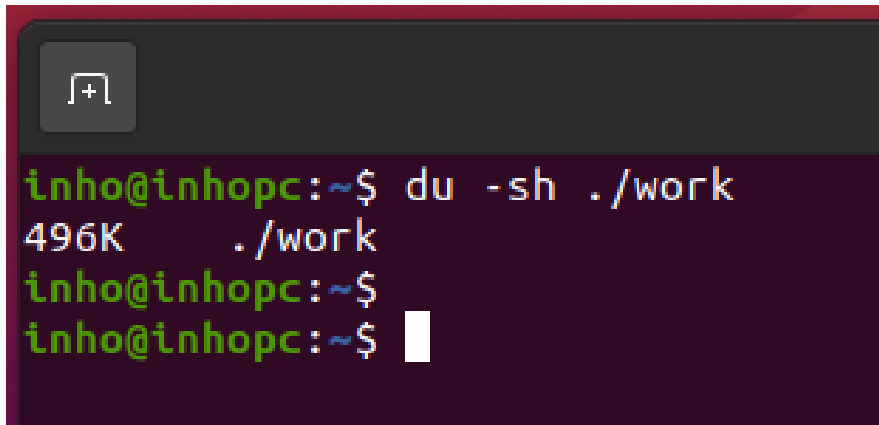
work 디렉토리 용량 확인 방법

- ls 명령어로 디렉토리 용량을 확인할 수 없다.
- 아래와 같이 확인하면
디렉토리는 무조건 4KB로 출력된다.

```
inho@inhopc:~$ ls -alh | grep work
drwxrwxr-x  4 inho inho 4.0K  3월 13 23:58 work
inho@inhopc:~$
```

du -sh [파일명]

- 디렉토리 or 파일의 용량을 확인한다.
- -s 옵션 = 디렉토리 개별이 아닌, 총 사용량만을 출력함
- -h 옵션 = 인간 옵션
 - 사람이 보기 쉽게 출력된다.

A terminal window with a dark background and a title bar. The title bar has a close button icon. The terminal shows the command 'du -sh ./work' being executed, followed by the output '496K ./work'. The prompt 'inho@inhopc:~\$' is visible before and after the command and output.

```
inho@inhopc:~$ du -sh ./work
496K    ./work
inho@inhopc:~$
inho@inhopc:~$
```

file 명령어

- 어떤 종류의 파일인지 확인하기
- 파일 or 디렉토리 or 실행파일 등 구분이 가능하다.

1. /dev/stdout
2. /etc/passwd
3. /proc/fs
4. /bin/zip
5. /dev/mem

```
inho@inhopc:~$  
inho@inhopc:~$ file work  
work: directory  
inho@inhopc:~$  
inho@inhopc:~$ file pink.c  
pink.c: C source, ASCII text  
inho@inhopc:~$  
inho@inhopc:~$ file yello.o  
yello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped  
inho@inhopc:~$  
inho@inhopc:~$
```

명령어 위치를 확인하고 싶을 때, **which**

- `which ls`
- `which reboot`
- `which grep`

Binary 파일

- 리눅스에서는 실행파일보다 Binary 파일 이라는 말을 더 많이 사용한다.
- `/usr/bin` 에 리눅스 Binary 파일이 모여있다.
 - Quiz. `/bin`과 `/usr/bin` 의 차이가 무엇일까?

패키지 설치하는 경우,
/usr에 설치된다.

- apt 패키지설치도구로 “sl” 을 설치해보자.
- sl 어느 위치에 설치되었는지 확인해보자.
- file로 파일 정보를 읽어보자.

다음 명령어를 수행해보자.

1. `find` 명령어로 “reboot”이 어디에 있는지 찾아내자
 - `type f` 옵션으로 파일만 찾자.
2. `which` 명령어로 “reboot” 명령어 위치를 찾자
 - 찾은 위치로 `ls -al` 로 확인해보자.
3. `file` 명령어로 “ls” 바이너리 파일 형태를 읽어보자.
 - ELF 를 확인하자.
4. `find` 명령어로 `/message` 디렉토리를 찾아내자
5. `/dev/cdrom` 은 심볼릭 링크이다.
이 원본의 위치를 찾아 `file` 명령어로 정보를 읽어내자.

다음 명령어를 수행해보자.

1. `find` 명령어로 “reboot”이 어디에 있는지 찾아내자
 - `type f` 옵션으로 파일만 찾자.
2. `which` 명령어로 “reboot” 명령어 위치를 찾자
 - 찾은 위치로 `ls -al` 로 확인해보자.
3. `file` 명령어로 “ls” 바이너리 파일 형태를 읽어보자.
 - ELF 를 확인하자.
4. `find` 명령어로 `/message` 디렉토리를 찾아내자
5. `/dev/cdrom` 은 심볼릭 링크이다.
이 원본의 위치를 찾아 `file` 명령어로 정보를 읽어내자.

내일 방송에서 만나요!

삼성 청년 SW 아카데미