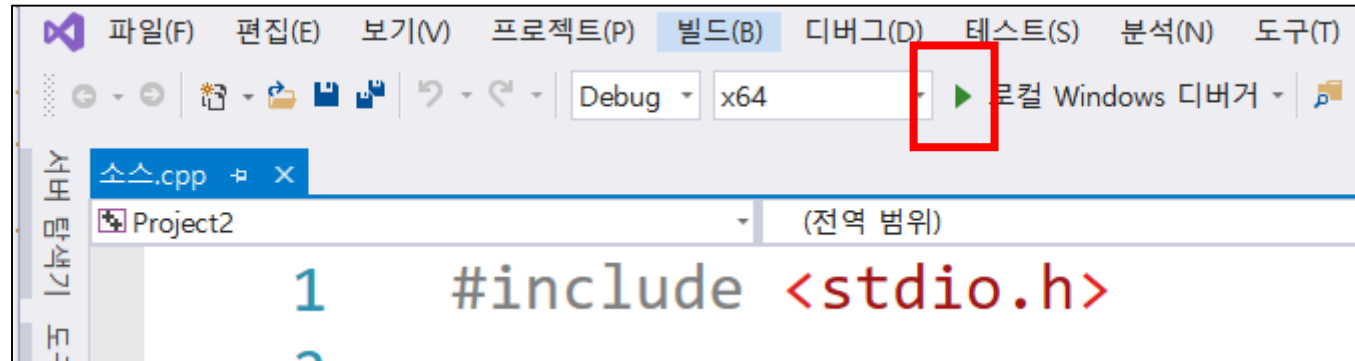


gcc Build Process

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 파일들을 하나로 합친다.
 - gcc **./green.o ./yellow.o** -o **./go**
 - -o 옵션 : output 파일 지정



green.o / yellow.o

각각의 Object 파일 (실행불가)

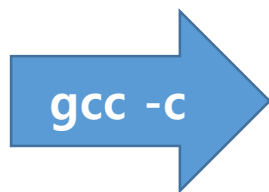
Linking



./go

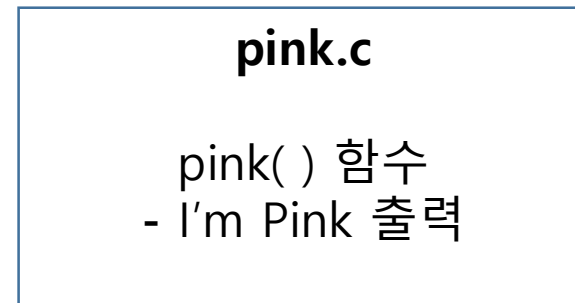
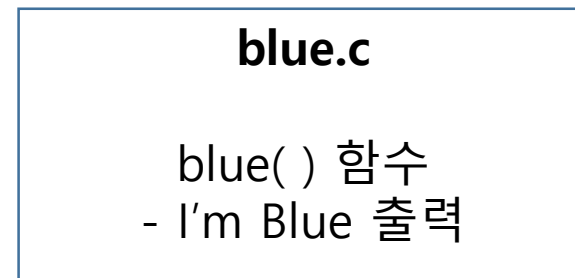
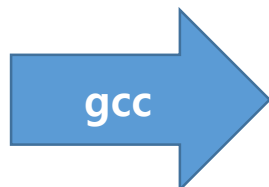
하나로 합쳐, 하나의 프로그램 생성
(실행 가능)

1. Compile & Assemble 하기



2. Linking 하기

- 실행파일 이름 : **bluepink**



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

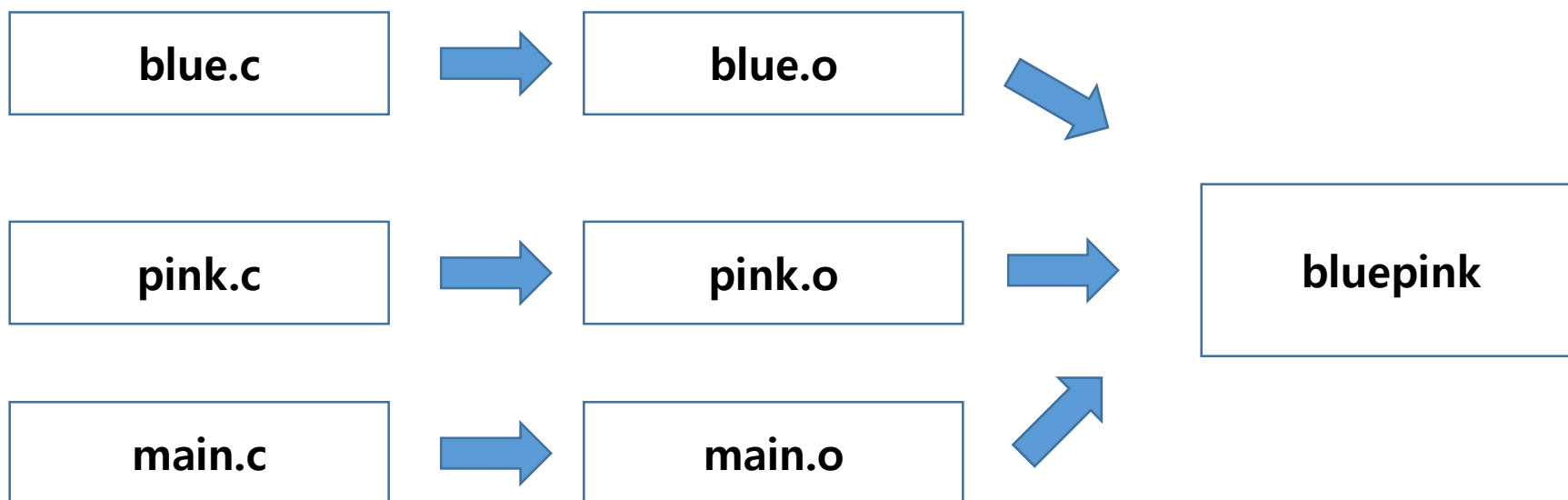
```
inho@inho:~/work$ gcc ./*.c
inho@inho:~/work$ ./a.out
I'm BLUE
I'm Pink
inho@inho:~/work$
```


빌드 자동화 스크립트

build script 제작하기

- 파일명 : **build.sh**

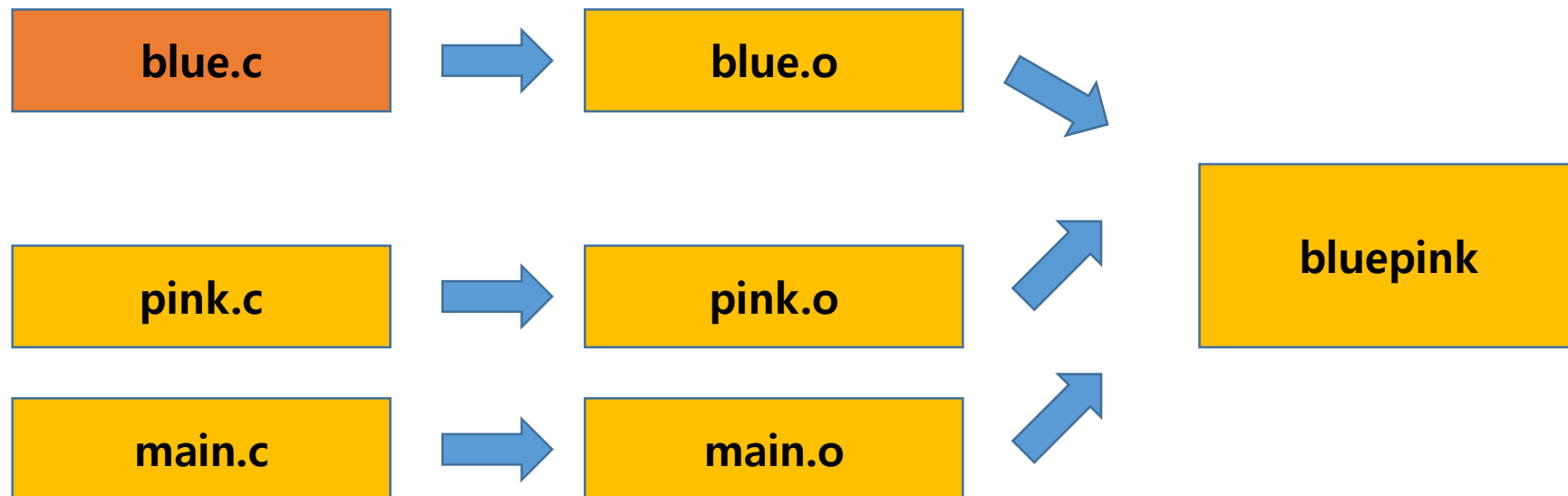
- gcc -c ./blue.c
- gcc -c ./pink.c
- gcc -c ./main.c
- gcc ./main.o ./blue.o ./pink.o -o ./bluepink



blue.c 파일 하나를 수정하였고, 테스트한다면?

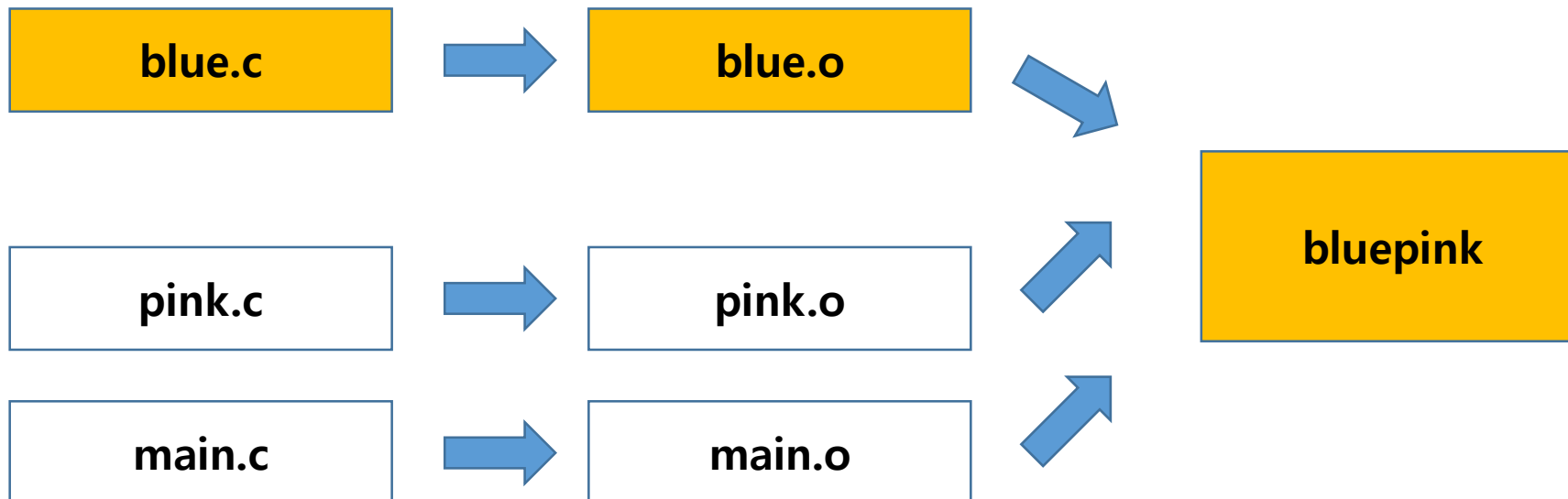
- build.sh 을 다시 수행해주면 됨

→ 모든 파일을 **Compile & Assemble** 수행함
(기존에 만들었던 모든 Object 파일들이 삭제되고 다시 생성됨)



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

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 : `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
- make를 한번 더 해본다. → 빌드 할 필요가 없다고 뜬다.
- blue.c 파일만 가볍게 수정 후 make를 해본다. →

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

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

모든 파일이
Compile & Assemble, Linking
되었음

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

필요한 파일만
Compile & Assemble, Linking
되었음

Make Build System의 장점 두 가지

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

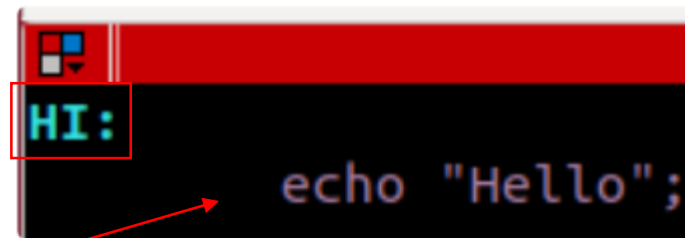
Make가 아래 방법보다 더 좋은 이유가 뭘까?

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

make 스크립트 시작

Hello World 출력하기

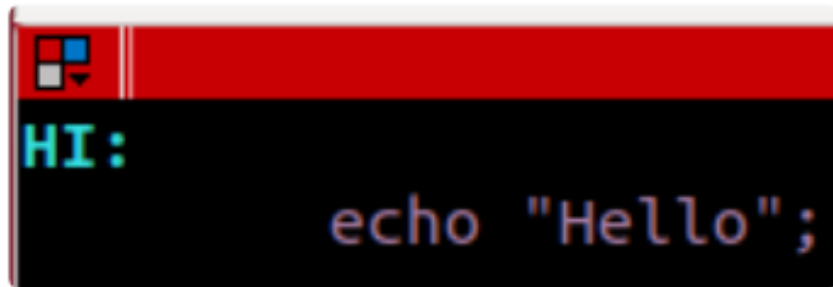
- Target이 반드시 1 개 이상 존재해야 함
- echo : 화면 출력 shell 명령어



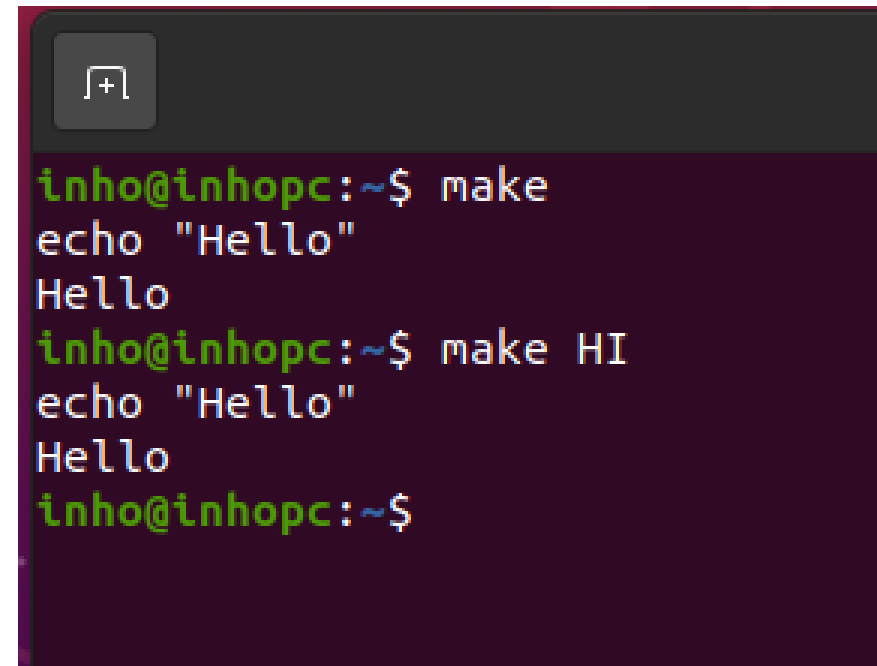
탭키 필수
(띄어쓰기 불가)

실행방법 두 가지

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



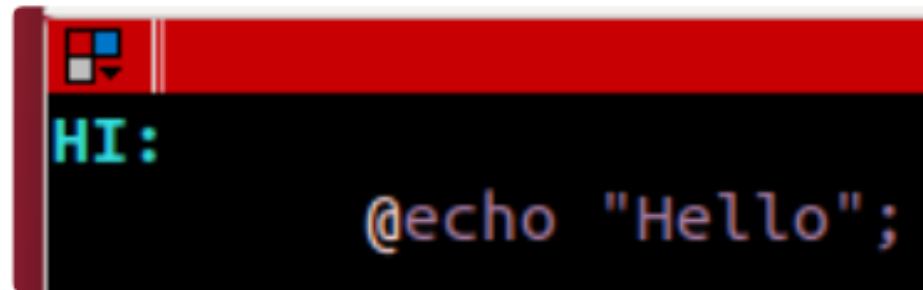
```
HI:  
    echo "Hello";
```



```
inho@inhopc:~$ make  
echo "Hello"  
Hello  
inho@inhopc:~$ make HI  
echo "Hello"  
Hello  
inho@inhopc:~$
```

Shell Script 명령어 @

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

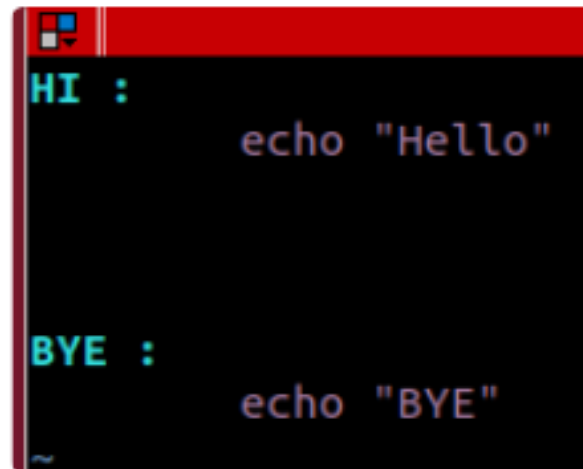


```
HI:
    @echo "Hello";
```

A terminal window with a red title bar and a black background. The prompt 'HI:' is shown in cyan. The command '@echo "Hello";' is entered in white text.

HI 와 BYE Target 만들기

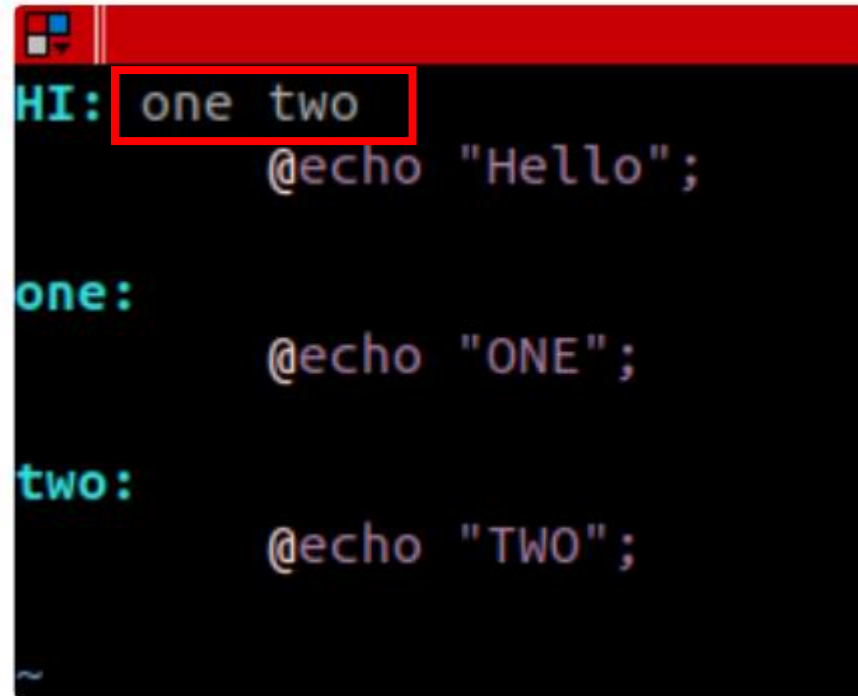
- 각각 수행해보자



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

의존성 타겟

- 수행 순서를 먼저 예측 후, 실습 진행하기



```
HI: one two
    @echo "Hello";

one:
    @echo "ONE";

two:
    @echo "TWO";

~
```

매크로는 글자 그대로 치환된다.

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

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

HI: one two
    @echo $(MSG1);

one:
    @echo $(MSG2);

two:
    @echo $(NONONO);
```

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 명령어

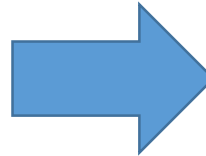
- `echo "HI" "ABC"`
 - HIABC 출력
- `echo "HI" "ABC"`
 - HI ABC 출력

```
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

```
inho@inhopc:~$ cat make
KFC="BTS"
HOHO="KFC"

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



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

make A 입력 시

- #으로 A 문자 만들어 출력하기

```
#  
# #  
###  
# #
```

make B 입력 시

- #으로 B 문자 만들어 출력하기

make All 입력 시

- A, B 문자 모두 출력하기

#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**
- 출력결과 : OH GOOD KFC

```
NAME = "OH"
NAME += "GOOD"
NAME += "KFC"

who:
    @echo $(NAME);
```

두 가지 대입 연산자

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

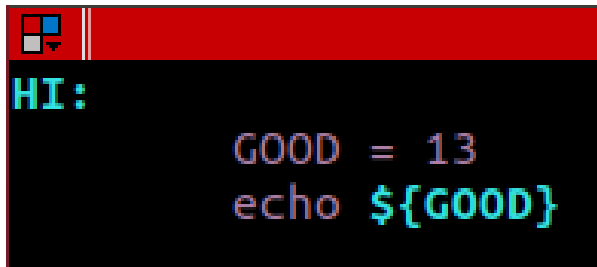
```
NAME = "OH"

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

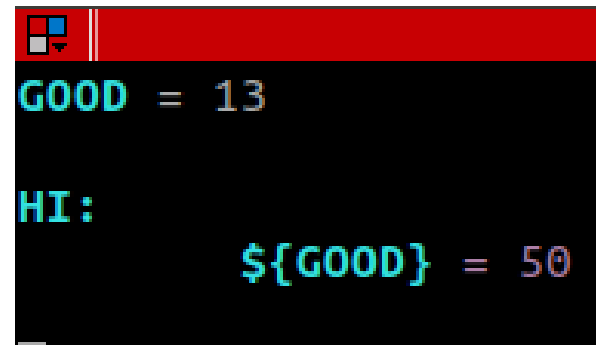
NAME += "GOOD"
NAME += "KFC"

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

Shell 명령어와 Make Script 를 구분하자



```
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

HI / HE 를 수행했을 때 출력 결과 예측하기

```
GOOD = 13

HI:
    echo ${GOOD}

GOOD = 55
GOOD := 20

HE:
    echo ${GOOD}
```

```
GOOD += 13

HI:
    echo ${GOOD}

GOOD += 55

HE:
    echo ${GOOD}
```

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

HI:
    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}

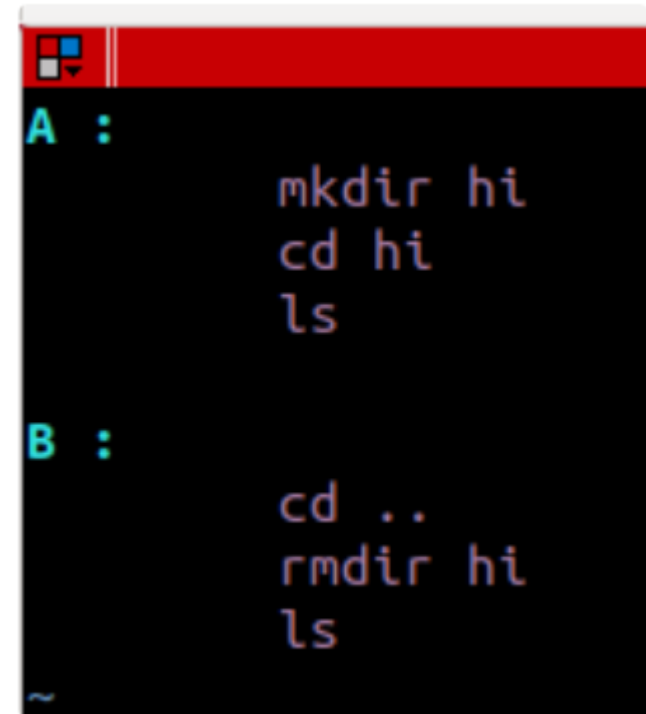
GOOD = "T"

HE:
    echo ${HOT}
```

자동화 스크립트 제작

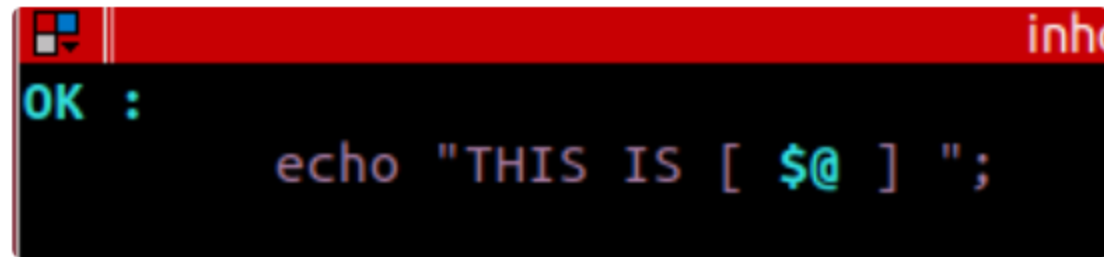
shell 명령어를 모아, 자동화 스크립트 제작 가능

- 순차적으로 명령을 수행한다.



```
A :  
    mkdir hi  
    cd hi  
    ls  
  
B :  
    cd ..  
    rmdir hi  
    ls  
~
```

`$@` = Target을 나타내는 변수

A terminal window with a red title bar. The text 'OK :' is on the first line, and 'echo "THIS IS [\$@] ";' is on the second line.

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

GCC MakeFile

다음 소스코드를 준비한다.

- go.c : 메인 함수
- hi.c / hi.h : 출력 함수

```
go.c + (~/jason) - VIM
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
1 #include <stdio.h>
2 #include "hi.h"
3
4 int main()
5 {
6     hello(1, 2, 3);
7
8     return 0;
9 }
```

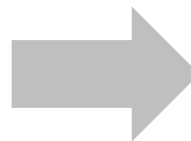
```
hi.h (~/jason) - VIM
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
1 #include <stdio.h>
2
3 void hello(int a, int b, int c);
~
~
```

```
hi.c + (~/jason) - VIM
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
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

```
Makefile (~/jason) - VIM
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
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

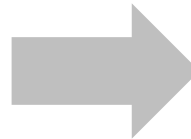
- 실제 임베디드 환경에서는 빌드해야하는 툴이 gcc가 아니라 다른 프로그램을 선택해야 할 때가 있다. (ARM ToolChain)

```
Makefile + (~/jason) - VIM
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
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
```

내장 매크로 사용

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

```
Makefile + (~/jason) - VIM
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
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
```



```
Mak
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T)
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 작업)

```
Makefile (~/jason) - VIM
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
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



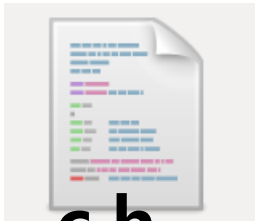
a.h

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



b.h

```
#define N1 11
```



c.h

```
#define N2 22
```

#include 역할

해당 파일을
그대로 복사 붙여넣기



```
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;
}
```

Build 과정 中
전처리작업 이후
변화

```
#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

```
#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;
```

```
}
```

```
~
```

```
~
```


오탈자 유의

```
incho@incho:~$ cat test2.c
#include "a.h"
#include "c.h"

void func1() {
    int n = N2;
    printf("TEST 2 : %d\n", n);
}
```

오탈자 유의

```
incho@incho  
#include "a.h"  
#include "b.h"  
#include "c.h"  
void func2() {  
    int n = N1 + N2;  
    printf("TEST 3 : %d\n", n);  
}  
~
```

1단계

- make 기본 문법대로 진행
- 빌드 테스트 해보기

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

```
inho@inhopc:~/work$ make
gcc -c test1.c
gcc -c test2.c
gcc -c test3.c
gcc -o result test1.o test2.o test3.o
inho@inhopc:~/work$
inho@inhopc:~/work$ ./result
TEST 1 : 11
TEST 2 : 22
TEST 3 : 33
inho@inhopc:~/work$
```

2단계

- **clean 추가**
 - 빌드 과정에 생기는 파일들 제거를 위함
- **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
- make
- make clean

테스트 2

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

테스트 3

- touch c.h
- make