출처: bowbowbow.tistory.com/12

make와 Makefile

---

make & Makefile 이란?

SHELL 에서 컴파일을 해보셨다면, make 명령어로 컴파일을 실행하는 경우를 자주 보셨을 것입니다. Makefile이 있는 디렉토리에서 make 만 치면 컴파일이 실행된다?? 어떻게 이런 일이 일어날 수 있는 것일까요?

왜냐하면 make는 **파일 관리 유틸리티** 이기 때문이지요.

make는

> 파일 간의 종속관계를 파악하여 **Makefile(기술파일)**에 적힌 대로 컴파일러에 명령하여 SHELL 명령이 순차적으로 실행될 수 있게 합니다.

그럼 이제 Makefile도 어떤 역할을 하는지 아시겠죠?

make를 쓰는 이유

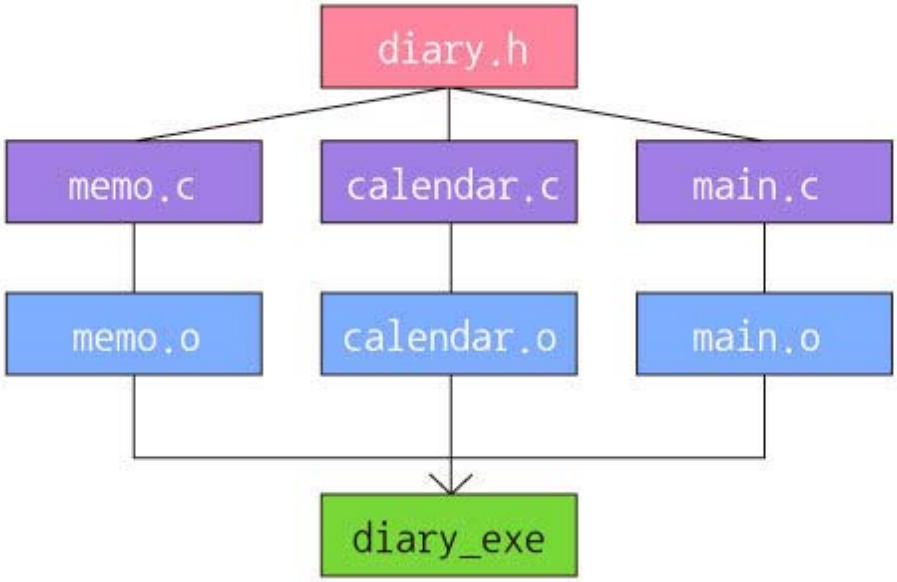
하지만 여기서 의문점이 있을 수 있습니다. 그냥 **컴파일러**로 **컴파일**하면 되지 왜 굳이 Makefile을 만들고 make명령을 실행해야 하나?

make을 쓰면 다음과 같은 장점이 있습니다.

1. 각 파일에 대한 반복적 명령의 자동화로 인한 시간 절약
2. 프로그램의 종속 구조를 빠르게 파악 할 수 있으며 관리가 용이
3. 단순 반복 작업 및 재작성을 최소화

글로만 보니 이해가 잘 안되시죠? 그럼 make의 필요성을 느껴보기 위해 **기본적인 컴파일** 과 **make를 이용한 컴파일**을 직접 해봅시다!

예제



이제 위의 종속관계 표를 보며 **diary_exe**라는 실행 파일을 만들어 봅시다!

1. diary.h 헤더파일 만들기

세 개의 c파일이 include 할 헤더파일을 생성해 봅시다!

vi diary.h (헤더 파일 생성)

코드

```
//diary.h

#include <stdio.h>
void memo();
void calendar();
```

2. 재료로 사용 될 C파일 만들기

vi memo.c
vi calendar.c
vi main.c

코드

1. memo.c

```c
//memo.c

#include "diary.h"
void memo(){

        printf("I'm function Memo! \n");
}
```

2. calendar.c

```c
//calendar.c

#include "diary.h"
void calendar(){
        printf("I'm function Calendar() \n");
}
```

3. main.c

```c
//main.c

#include "diary.h"


int main(void){

        memo();
        calendar();
        return 0;
}
```

3. 생성된 파일 확인하기

위의 모든 파일을 생성 했다면 제대로 생성 되었는지 **ls**명령어로 확인해 줍시다.

$ ls

```
ubuntu@ip-172-31-10-167:~/pintos/makeEx/exMake$ ls
calendar.c  diary.h  main.c  memo.c
ubuntu@ip-172-31-10-167:~/pintos/makeEx/exMake$
```

자! 모든 파일을 생성했다면 이제 두가지 방법으로 컴파일을 해보겠습니다.

기본적인 컴파일 과정

먼저 기본적인 방법으로 컴파일을 해봅시다. 컴파일은 gcc 를 이용하였습니다.

1. c파일에서 object 파일 생성하기

아래의 명령어로

```
gcc -c -o memo.o memo.c
gcc -c -o calendar.o calendar.c
gcc -c -o main.o main.c
```

각 c파일에서 object 파일을 생성해 줍니다.

여기서 -c 옵션은 object 파일을 생성하는 옵션이고,
-o 옵션은 생성 될 파일 이름을 지정하는 옵션입니다.

> 여기서는 -o 옵션을 넣지 않아도 object 파일이름이 **(c파일이름).o** 로 자동 생성 됩니다.
> 하지만 실행 파일 생성시 -o 옵션을 넣지 않으면 모든 파일이 **a.out** 이라는 이름을 가지게 되므로 여러 개의 실행 파일을 생성해야 할 때 효율적인 옵션입니다.

그럼
ls 명령어로 **object 파일**이 제대로 생성되었는지 확인해 줍시다.

```
ubuntu@ip-172-31-10-167:~/pintos/makeEx/exMake$ gcc -c main.c
ubuntu@ip-172-31-10-167:~/pintos/makeEx/exMake$ gcc -c memo.c
ubuntu@ip-172-31-10-167:~/pintos/makeEx/exMake$ gcc -c calendar.c
ubuntu@ip-172-31-10-167:~/pintos/makeEx/exMake$ ls
calendar.c  calendar.o  diary.h  main.c  main.o  memo.c  memo.o
ubuntu@ip-172-31-10-167:~/pintos/makeEx/exMake$
```

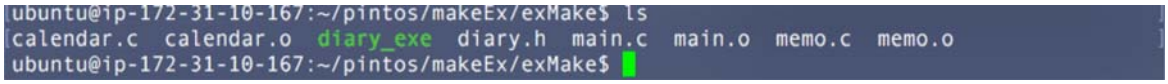
2. 각 object파일을 묶어 컴파일을 통해 diary_exe 실행파일 생성하기

이제 실행 파일을 생성해 봅시다!

```
gcc -o diary_exe main.o memo.o calendar.o
```

여기서 object 파일들의 순서는 상관이 없습니다.
위의 명령어를 실행하면 드디어 **diary_exe** 실행파일이 생성됩니다!!!!

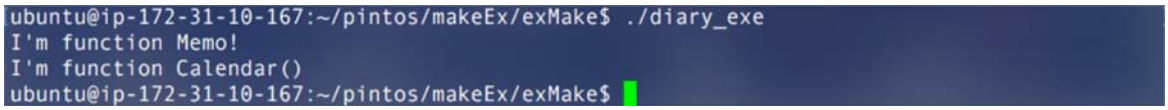
ls 명령어로 확인해 봅시다.



3. 결과 확인하기

바르게 생성되었다면 아래와 같이 결과가 나오는지 확인해 보세요.

> $./diary_exe



> 기존의 컴파일 과정이 여기서는 그리 귀찮지 않습니다. 모든 c파일을 각각 컴파일 해도 3번만 명령해 주면 되니까요. 하지만 만약 하나의 실행파일을 생성하는데 필요한 c파일이 **1000개**라면..?? **1000개**의 명령어가 필요합니다. 이러한 상황을 해결해 주는 것이 바로 `make` 와 `Makefile`입니다!

make를 이용한 컴파일 과정

그럼 이제 `Makefile` 을 먼저 어떻게 만드는지 알아 본 후 `make` 명령으로 위의 파일들을 컴파일 해봅시다.

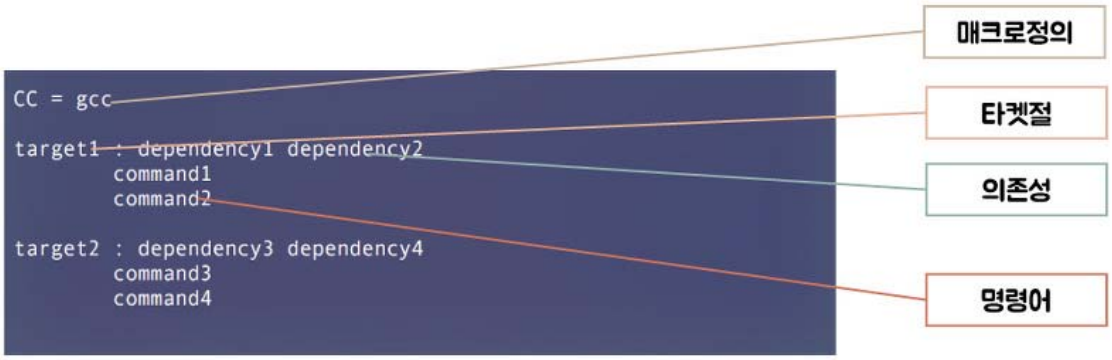
Makefile 의 구성

`Makefile`은 다음과 같은 구조를 가집니다.

> -목적파일(Target) : 명령어가 수행되어 나온 결과를 저장할 파일
> -의존성(Dependency) : 목적파일을 만들기 위해 필요한 재료
> -명령어(Command) : 실행 되어야 할 명령어들
> -매크로(macro) : 코드를 단순화 시키기 위한 방법

Makefile의 기본구조

위의 구성에서 말한 요소들은 실제 Makefile 코드에서 다음과 같이 배치됩니다.



Makefile 작성규칙

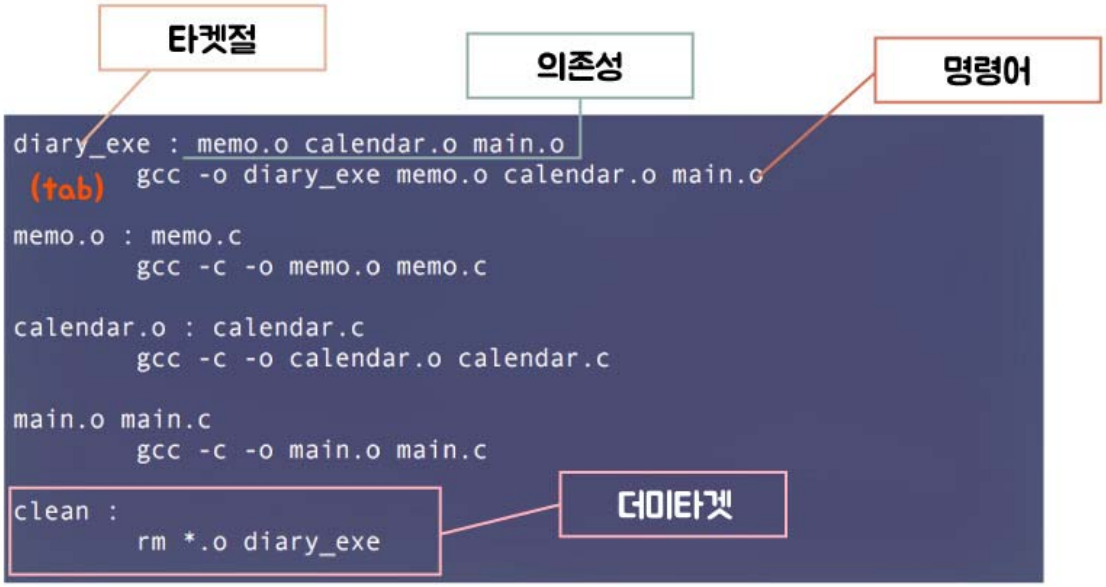
목표파일 : 목표파일을 만드는데 필요한 **구성요소들**
(tab)목표를 달성하기 위한 명령 1
(tab)목표를 달성하기 위한 명령 2

// 매크로 정의 : Makefile에 정의한 string 으로 치환한다.
// 명령어의 시작은 반드시 **탭**으로 시작한다.
// Dependency가없는 target도 사용 가능하다.

make 예제 따라해보기

자! 이제 실제로 Makefile을 만들어 봅시다~

```
$ vi Makefile
```



여기서 **더미타겟** 은 파일을 생성하지 않는 개념적인 타겟으로

```
$ make clean
```

라 명령하면 현재 디렉토리의 모든 **object 파일**들과 생성된 실행 파일인 **diary_exe**를 rm 명령어로 제거해 줍니다.

이제

```
$ make
```

로 Makefile을 실행해 줍니다.

```
ubuntu@ip-172-31-10-167:~/fsLab/exMake$ ls
calendar.c  diary.h  main.c  Makefile  memo.c
ubuntu@ip-172-31-10-167:~/fsLab/exMake$ make
gcc    -c -o memo.o memo.c
gcc    -c -o main.o main.c
gcc    -c -o calendar.o calendar.c
gcc  -o diary_exe memo.o main.o calendar.o
ubuntu@ip-172-31-10-167:~/fsLab/exMake$ ls
calendar.c  diary_exe  main.c  Makefile  memo.o
calendar.o  diary.h    main.o  memo.c
ubuntu@ip-172-31-10-167:~/fsLab/exMake$ ./diary_exe
Function Memo .
Function Calendar()
ubuntu@ip-172-31-10-167:~/fsLab/exMake$
```

명령어들이 실행 되면서 타겟파일이였던 **diary_exe** 가 만들어졌습니다!!

실행결과는 기본적인 컴파일 과정에서 본 결과와 동일함을 알 수

있습니다.

그런데 아직까지 기본적인 컴파일 과정을 묶어둔 것 외에는 특별한 점이 없어 보입니다.

> 위의 코드를 더욱 단순화 시키기 위해서 **매크로(macro)**를 사용해 봐요~

Makefile 개선하기 : 매크로 사용

매크로는 생각보다 간단합니다. 위의 코드에서 **중복되는 파일 이름들을 특정 단어로 치환**하면 됩니다.

마치 C언어에서 **#define**을 하는 것과 비슷한 원리입니다.

Makefile 매크로 사용 예제

```
CC = gcc
CFLAGS = -W -WALL
TARGET = diary_exe

$(TARGET) : memo.o calendar.o main.o
	$(CC) $(CFLAGS) -o $(TARGET) memo.o calendar.o main.o

memo.o : memo.c
	$(CC) $(CFLAGS) -c -o memo.o memo.c

calendar.o : calendar.c
	$(CC) $(CFLAGS) -c -o calendar.o calendar.c

main.o main.c
	$(CC) $(CFLAGS) -c -o main.o main.c

clean :
	rm *.o diary_exe
```

작성 규칙

1. 매크로를 참조 할 때는 소괄호나 중괄호 둘러싸고 앞에 '$'를 붙인다.
2. 탭으로 시작해서는 안되고 , :,=,#,"" 등은 매크로 이름에 사용할 수 없다.
3. 매크로는 반드시 치환될 위치보다 먼저 정의 되어야 한다.

여기서 **-W -Wall**는 컴파일 시 컴파일이 되지 않을 정도의 오류라도 모두 출력되게 하는 옵션입니다.

> make clean
> vi Makefile //매크로 사용 예제처럼 수정
> ./diary_exe

로 전과 같은 결과가 나오는지 확인해 보세요~

여기서 더 코드를 단순화 시키기 위해서 사용자가 직접 정의하는 매크로가 아닌 미리 정의된 **내부 매크로**를 한번 사용해 보겠습니다!

Makefile 개선하기2 : 내부 매크로 사용

```
CC = gcc
CFAGS = -W -Wall
TARGET = diary_exe
OBJECTS = memo.o main.o calendar.o

all : $(TARGET)

$(TARGET): $(OBJECTS)
        $(CC) $(CFLAGS) -o $@ $^

clean :
        rm *.o diary_exe
```

!?
내부 매크로를 사용하였더니 코드가 굉장히 단순해 졌습니다!
여기서 사용된 내부 매크로를 한번 살펴봅시다.

1. “$@” : 현재 타겟의 이름
2. “$^” : 현재 타겟의 종속 항목 리스트

이를 바탕으로 위의 코드를 한번 처음부터 끝까지 해석해 봅시다!

> **1. gcc 컴파일러를 이용**
> **2. 사소한 오류까지 출력**
> **3. 최종 타겟 파일은 diary_exe**
> **4. OBJECT 로 정의할 파일들은 memo.o main.o calendar.o**

5. all 은 현재는 사용하지 않았지만 타겟 파일이 여러개 일때 사용됩니다.
6. 타겟 파일을 만들기 위해 OBJECT 들을 사용한다.(단 OBJECT 파일이 없다면 OBJECT 파일과 이름이 동일한 C파일을 찾아 OBJECT파일을 생성한다.)
7. gcc -o diary_exe memo.o main.o calendar.o과 동일
8. 더미타겟

이해가 되셨나요?

내부 매크로는 본 예제에서 쓰인 것 보다 훨씬 많기 때문에 리스트를 한번 찾아보고 다른 예제를 해보시면 도움이 됩니다 :-)

정리

이처럼 Makefile을 생성하여 make 명령을 사용하면 다음과 같은 장점이 있습니다.

- 입력파일 변경 시 결과파일 자동 변경을 원할 때 지능적인 배치작업 수행
- 일일이 gcc 명령어를 안치고도 간단하면서 용이하게 컴파일을 진행할 수 있음

우리 모두 make 로 더 쉽게 컴파일 해요.

끝