Background

Makefile

Macro for Makefile

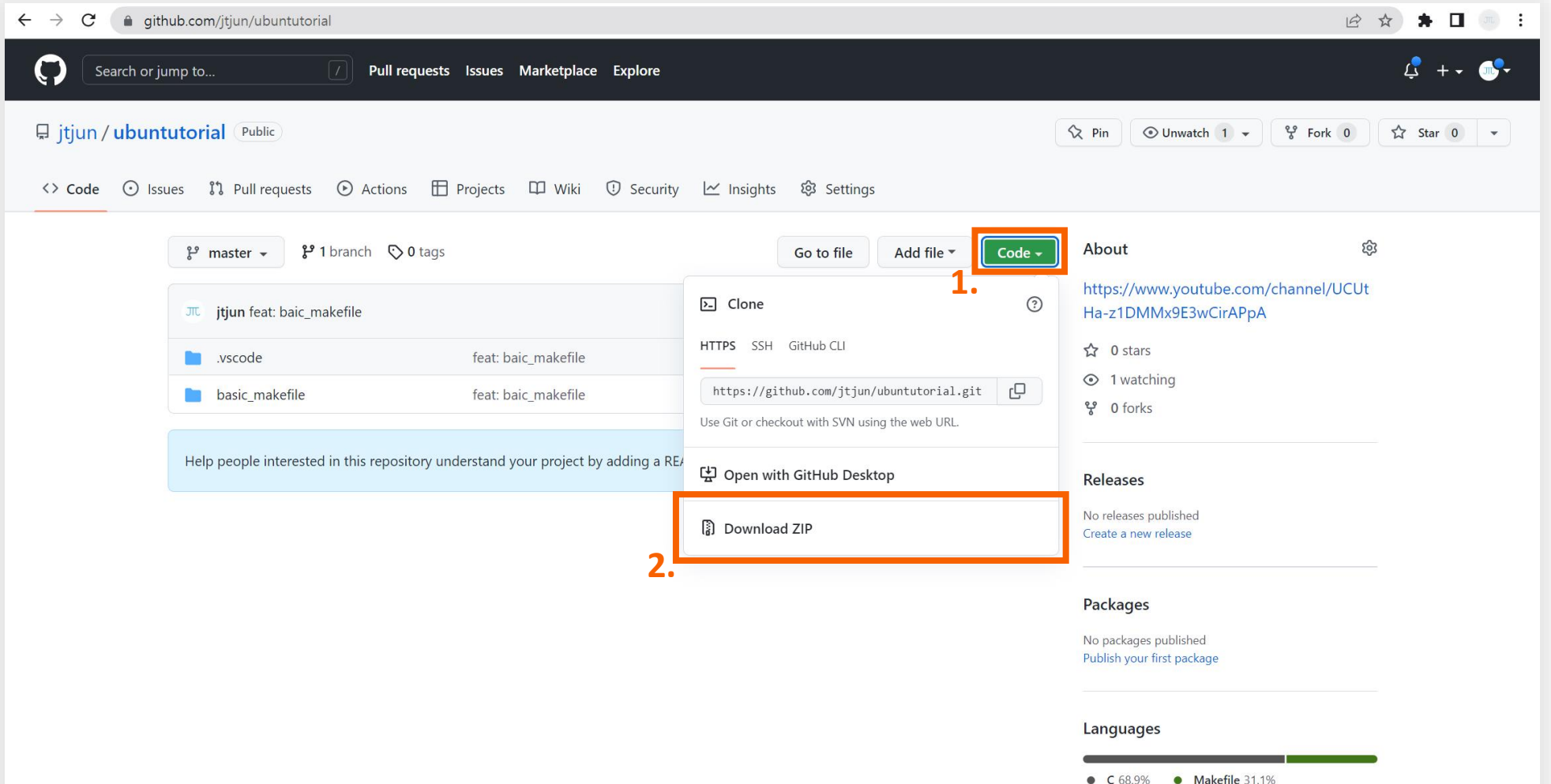
Clean-Up

Basic Makefile

tinyurl.com/ssdlabMakefile

Preparation

- 실습 파일 : https://github.com/jtjun/ubuntutorial



Example Codes

main.c

```c
#include "foo.h"
#include "bar.h"

int main() {
  int x = foo(bar());
  printf("result is %d\n", x);
  return 0;
}
```

foo.h

```c
#include <stdio.h>

int foo(int x);
```

bar.h

```c
#include <stdio.h>

int bar();
```

foo.c

```c
#include "foo.h"

int foo(int x) {
  printf("foo returns %d + 1\n", x);
  return x + 1;
}
```

bar.c

```c
#include "bar.h"

int bar() {
  printf("bar returns 2\n");
  return 2;
}
```

Compile & Linking ?

Makefile ?

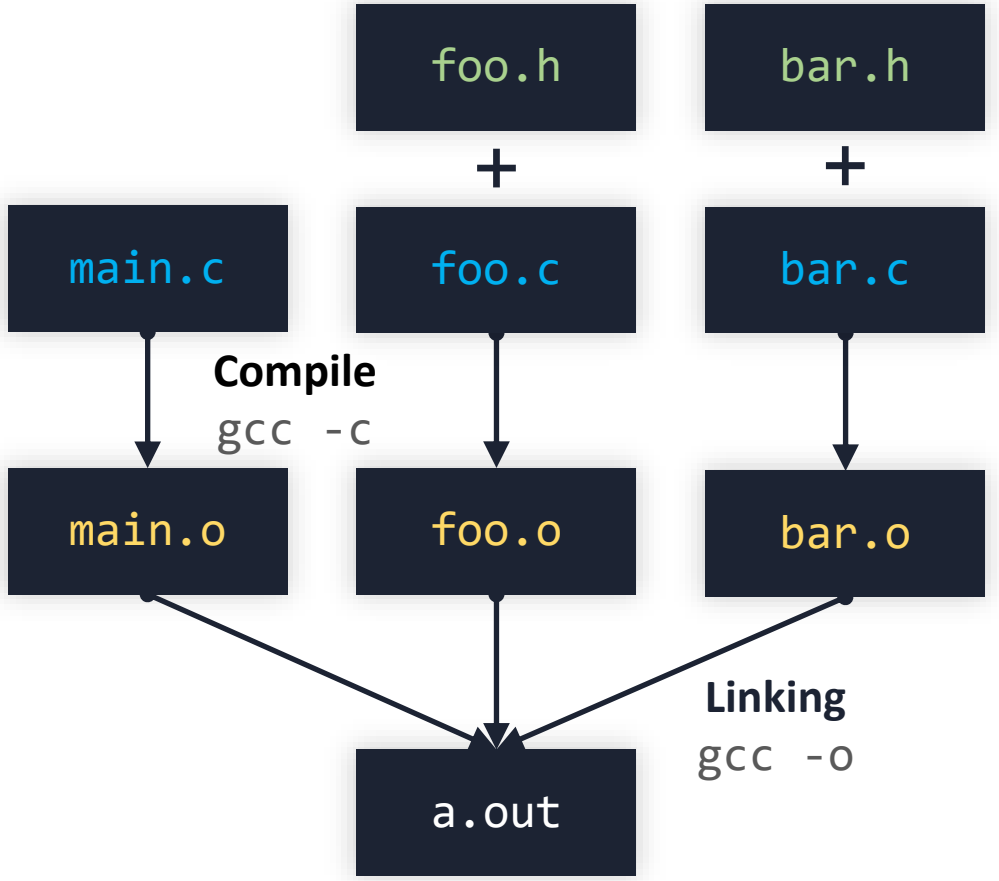
‘make’의 필요성

‘make’ 옵션

Background

Compile & Linking ?

- **Compile**
  - 소스코드 → 기계어
  - 소스파일(.c)로 목적파일(.o) 생성
- **Linking**
  - 여러 소스코드 → 실행파일
  - 서로 다른 소스파일을 묶어,
    하나의 실행파일 생성



Makefile ?

- **GNU make 프로그램의 설정파일**
  - 실행파일을 생성하기 위해 반복적으로 수행하는 **Compile**과 **Linking**을 쉽게 하기 위해 사용
  - 무엇을 새로 컴파일해야 하는지 자동으로 판단, 필요한 커맨드를 이용해 **그것만 재 컴파일** (Incremental Build)
  - 라이브러리 및 컴파일 환경을 관리
  - Makefile을 makefile database라 하기도 함
    파일명: Makefile (makefile, GNUmakefile)

‘make’의 필요성

- **make를 사용하지 않을 경우,**
  - 개별 파일 컴파일 후,
    최종 실행파일 생성
  - 파일이 많아질수록 명령어 증가
    → 쉘 스크립트를 쓸까?
  - 쉘 스크립트를 쓴다면,
    하나의 파일만 수정해도
    모든 명령어를 다시 수행함
    → make를 쓰자!

```
$ gcc -c main.c↲
$ gcc -c foo.c↲
$ gcc -c bar.c↲
$ gcc -o a.out main.o foo.o bar.o↲
$ ./a.out↲
bar returns 2
foo returns 2 + 1
Result is 3
$
```

‘make’ 옵션

- -h : help
  도움말 출력
- -p : print database
  ‘make’내부에 세팅 되어 있는 값 출력
- -k : keep going
  에러가 나도 계속 실행
- -r : no built-in rules
  내장된 규칙 무시
- -d : debugging
  디버깅 관련 모든 정보 출력
- -f <FILE> : 인자로 받은 <FILE>을 Makefile로 취급

Makefile 문법

Basic Makefile

Incremental Build

Makefile

Makefile 문법

```
<macro>

<target>: <dependencies>
    <command>
```

Tab 으로 들여쓰기
(missing separator. Stop. 오류)

macro를 처리한 뒤, **dependencies**를 가지고 **command**를 수행하여 **target**을 생성함.

- **<macro>**
  #define처럼 키워드를 대체하는 매크로
- **<target>**
  빌드 대상, 명령어가 수행된 최종 결과
- **<dependencies>**
  <target>을 make할 때 필요한 파일 목록
  (Incremental Build가 가능하게 함)
- **<command>**
  <target>을 make하기 위해 실행할 명령어

Basic Makefile

```
a.out: main.o foo.o bar.o
    gcc -o a.out main.o foo.o bar.o

main.o: foo.h bar.h main.c
    gcc -c main.c

foo.o: foo.h foo.c
    gcc -c foo.c

bar.o: bar.h bar.c
    gcc -c bar.c
```

- make 명령어로 빌드
- 파일 변경 없이 make 할 경우,
  아무 것도 실행되지 않음

```
$ make↲
gcc -c main.c
gcc -c foo.c
gcc -c bar.c
gcc -o a.out main.o foo.o bar.o
$ make↲
make: 'a.out' is up to date.
$
```

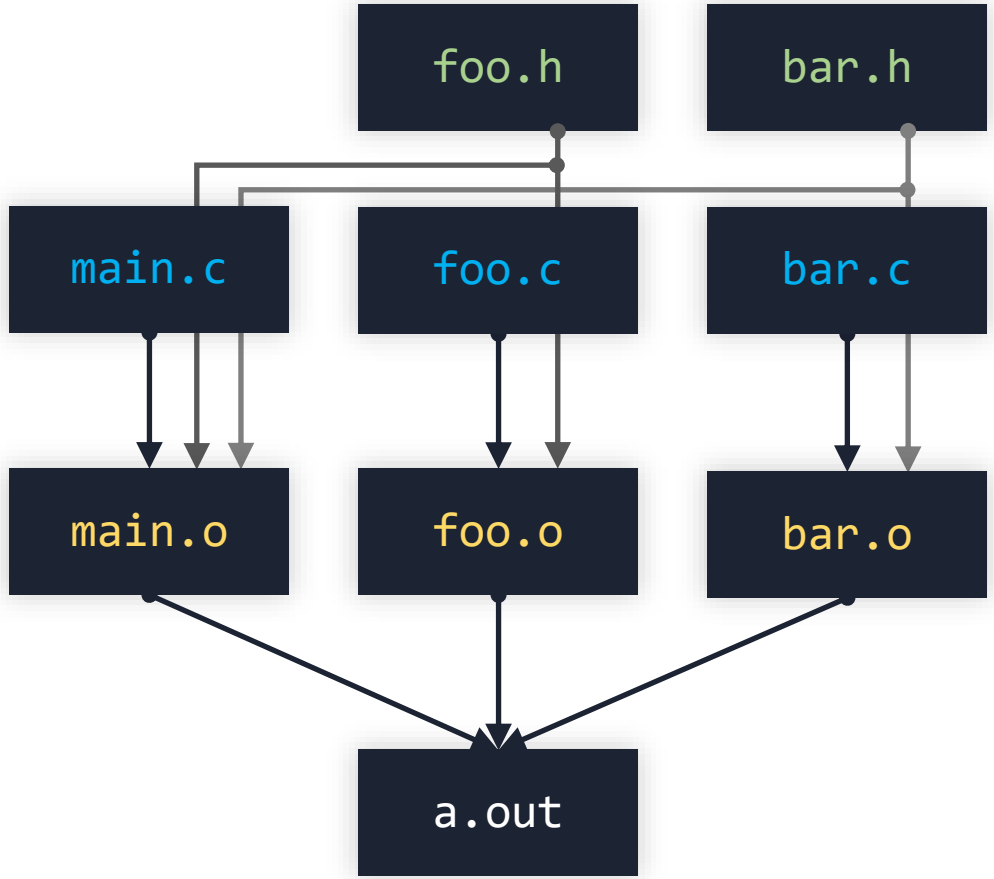
Incremental Build

```
a.out: main.o foo.o bar.o
    gcc -o a.out main.o foo.o bar.o

main.o: foo.h bar.h main.c
    gcc -c main.c

foo.o: foo.h foo.c
    gcc -c foo.c

bar.o: bar.h bar.c
    gcc -c bar.c
```



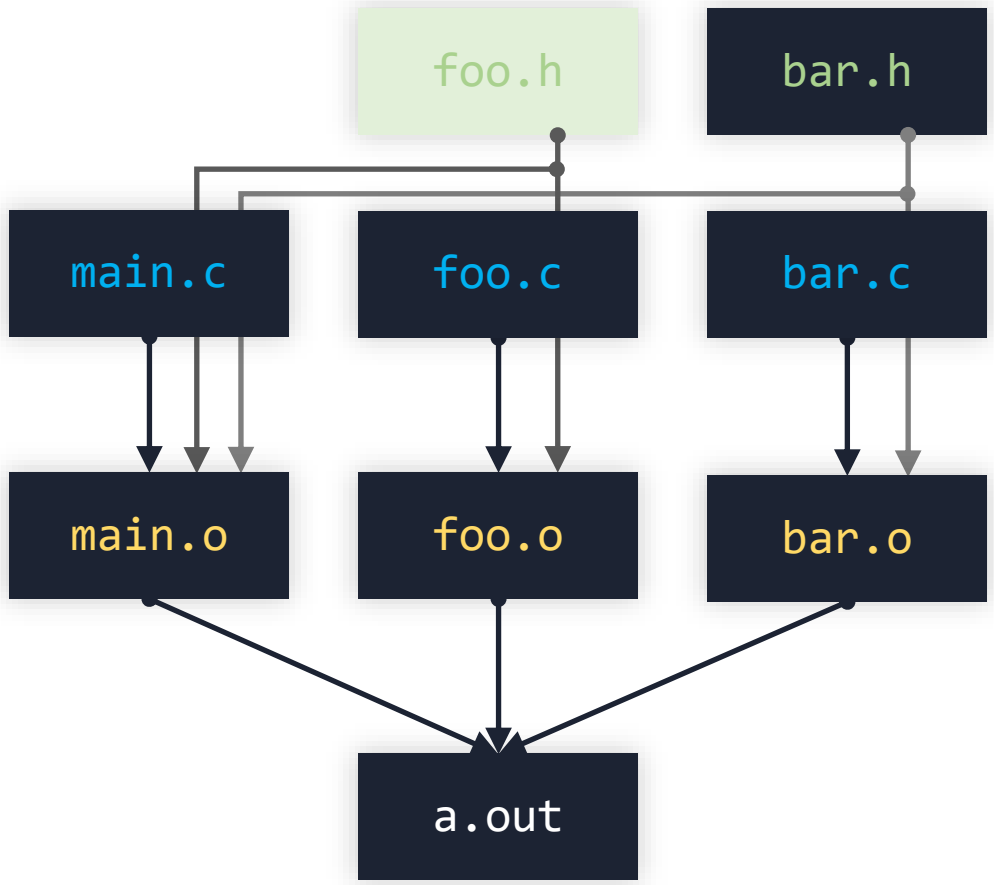
Incremental Build

```
a.out: main.o foo.o bar.o
    gcc -o a.out main.o foo.o bar.o

main.o: foo.h bar.h main.c
    gcc -c main.c

foo.o: foo.h foo.c
    gcc -c foo.c

bar.o: bar.h bar.c
    gcc -c bar.c
```



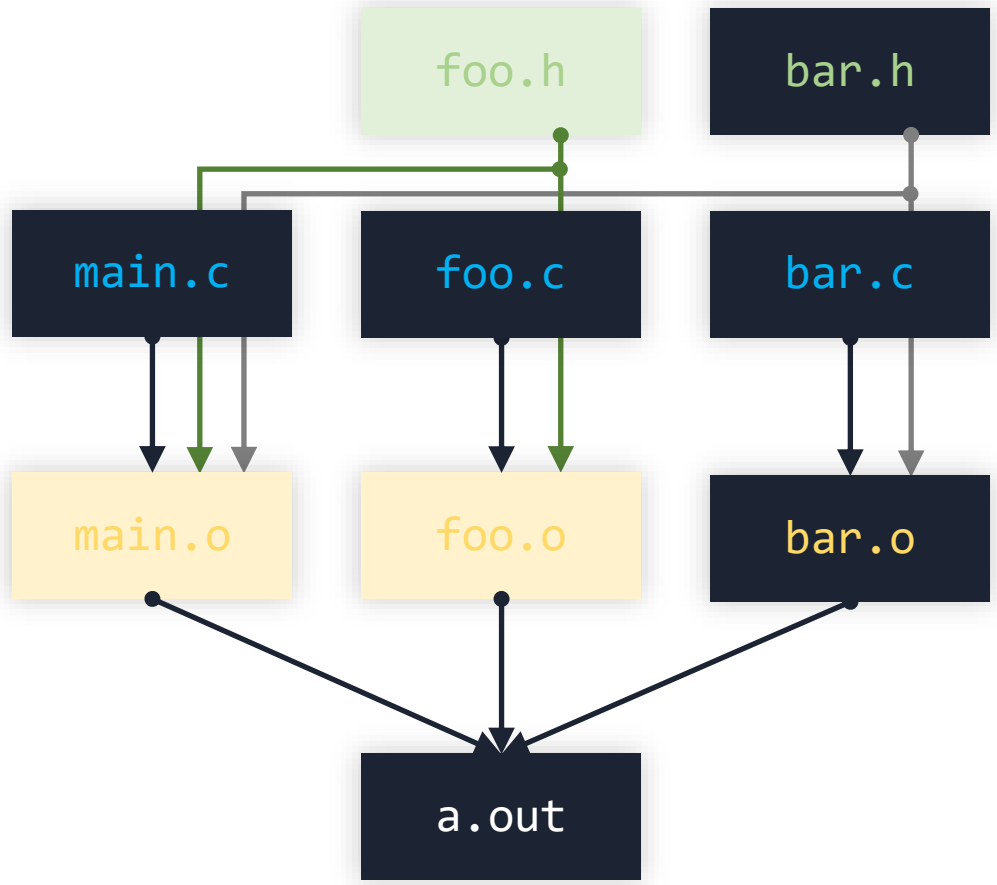
Incremental Build

```
a.out: main.o foo.o bar.o
    gcc -o a.out main.o foo.o bar.o

main.o: foo.h bar.h main.c
    gcc -c main.c

foo.o: foo.h foo.c
    gcc -c foo.c

bar.o: bar.h bar.c
    gcc -c bar.c
```



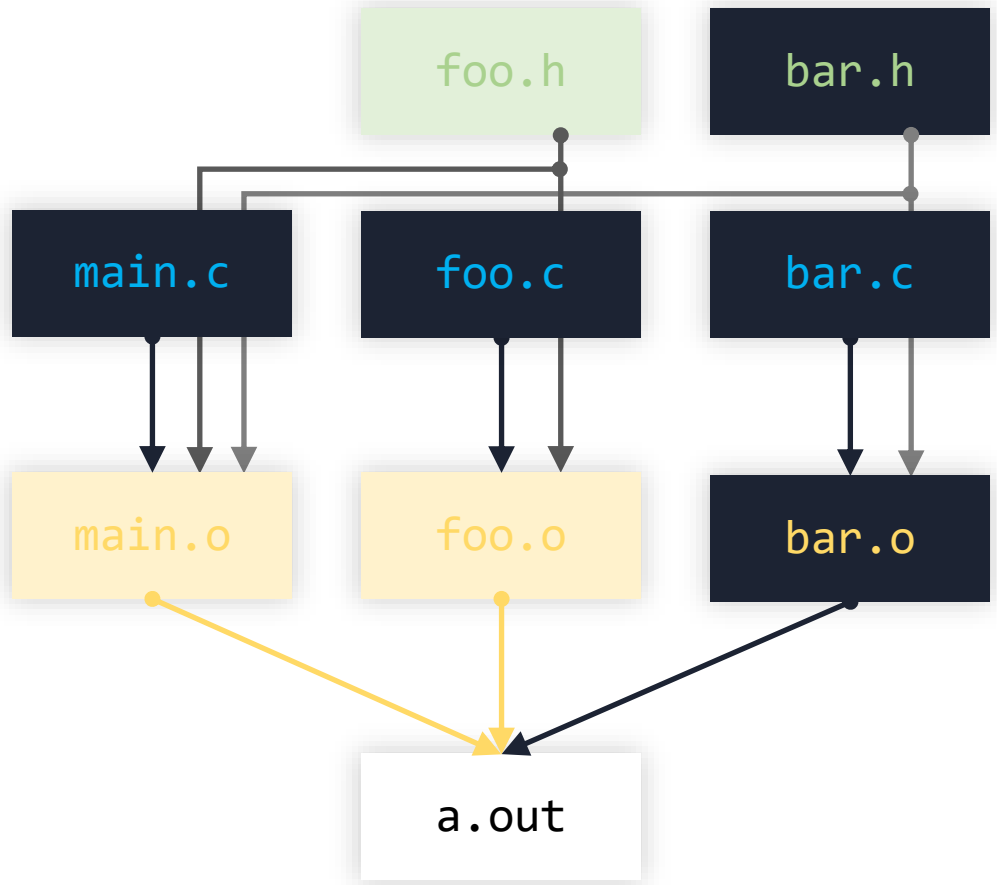
Incremental Build

```
a.out: main.o foo.o bar.o
    gcc -o a.out main.o foo.o bar.o

main.o: foo.h bar.h main.c
    gcc -c main.c

foo.o: foo.h foo.c
    gcc -c foo.c

bar.o: bar.h bar.c
    gcc -c bar.c
```



Incremental Build

```
a.out: main.o foo.o bar.o
    gcc -o a.out main.o foo.o bar.o

main.o: foo.h bar.h main.c
    gcc -c main.c

foo.o: foo.h foo.c
    gcc -c foo.c

bar.o: bar.h bar.c
    gcc -c bar.c
```

- 파일(foo.h) 변경 후 make 할 경우, Dependency가 있는 것만 실행

```
$ make↲
gcc -c main.c
gcc -c foo.c
gcc -o a.out main.o foo.o bar.o
$
```

- gcc -c bar.c 가 실행되지 않음 (변경한 파일(foo.h)과 의존 관계가 없음)

Makefile with Macro

최종 Makefile

Macro for Makefile

Makefile with Macro

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

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

main.o: foo.h bar.h main.c
    $(CC) -c main.c

foo.o: foo.h foo.c
    $(CC) -c foo.c

bar.o: bar.h bar.c
    $(CC) -c bar.c
```

- **$(MACRO)**
  - **CC** = <Compiler>
  - **TARGET** = <Build Target>
  - **OBJS** = <Object Files>
- **예약된 매크로**
  - CFLAGS: 컴파일 옵션 (-g: 디버그)
  - LDFLAGS: ld 옵션
  - LDLIBS: 링크 라이브러리

Makefile with Macro

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

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

$@      $^   $<
main.o: foo.h bar.h main.c
    $(CC) -c main.c

foo.o: foo.h foo.c
    $(CC) -c foo.c

bar.o: bar.h bar.c
    $(CC) -c bar.c
```

- **내부 매크로**
  - $@ : 현재 Target의 이름
  - $^ : Target 이 의존하는 대상들의 목록 (dependencies)
  - $< : dependencies 중에서 첫번째 파일 (가장 왼쪽)
  - $* : 현재 Target의 확장자를 제외한 이름
  - $? : dependencies 중에서 가장 최근에 변경된 파일

Makefile with Macro

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

all: $(TARGET)

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

.c.o:
    $(CC) -c -o $@ $<

clean:
    rm *.o a.out
```

- **all**
  최종적으로 만들 파일
- **확장자 규칙**
  .c.o : .c 를 컴파일 해서, .o 로 만듦
- **clean**
  - 빌드 중 발생한 부수적인 파일 정리
  - make clean 명령어로 실행

최종 Makefile

```
a.out: main.o foo.o bar.o
    gcc -o a.out main.o foo.o bar.o

main.o: foo.h bar.h main.c
    gcc -c main.c

foo.o: foo.h foo.c
    gcc -c foo.c

bar.o: bar.h bar.c
    gcc -c bar.c
```

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

all: $(TARGET)

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

.c.o:
    $(CC) -c -o $@ $<

clean:
    rm *.o a.out
```

Makefile 읽기

Clean-Up

Makefile 읽기

- gcc 컴파일러 사용
- 최종 타겟 파일은 a.out
- 매크로 OBJS 정의
- 컴파일 옵션 -g
  (디버그 정보 표시)
- 타겟 파일을 만들기 위해
  OBJS를 사용하여, 아래 명령어 실행
- .c 소스코드를 .o 목적파일로 컴파일
- clean 명령어 정의

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

all: $(TARGET)

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

.c.o:
    $(CC) -c -o $@ $<

clean:
    rm *.o a.out
```

Makefile 읽기

- gcc 컴파일러 사용
- 최종 타겟 파일은 a.out
- 매크로 OBJS 정의
- 컴파일 옵션 -g
  (디버그 정보 표시)
- 타겟 파일을 만들기 위해
  OBJS를 사용하여, 아래 명령어 실행
- .c 소스코드를 .o 목적파일로 컴파일
- clean 명령어 정의

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

all: $(TARGET)

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

.c.o:
    $(CC) -c -o $@ $<

clean:
    rm *.o a.out
```

Makefile 읽기

- gcc 컴파일러 사용
- 최종 타겟 파일은 a.out
- 매크로 OBJS 정의
- 컴파일 옵션 -g
  (디버그 정보 표시)
- 타겟 파일을 만들기 위해
  OBJS를 사용하여, 아래 명령어 실행
- .c 소스코드를 .o 목적파일로 컴파일
- clean 명령어 정의

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

all: $(TARGET)

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

.c.o:
    $(CC) -c -o $@ $<

clean:
    rm *.o a.out
```

Makefile 읽기

- gcc 컴파일러 사용
- 최종 타겟 파일은 a.out
- 매크로 OBJS 정의
- 컴파일 옵션 -g
  (디버그 정보 표시)
- 타겟 파일을 만들기 위해
  OBJS를 사용하여, 아래 명령어 실행
- .c 소스코드를 .o 목적파일로 컴파일
- clean 명령어 정의

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

all: $(TARGET)

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

.c.o:
    $(CC) -c -o $@ $<

clean:
    rm *.o a.out
```

Makefile 읽기

- gcc 컴파일러 사용
- 최종 타겟 파일은 a.out
- 매크로 OBJS 정의
- 컴파일 옵션 -g
  (디버그 정보 표시)
- 타겟 파일을 만들기 위해
  OBJS를 사용하여, 아래 명령어 실행
- .c 소스코드를 .o 목적파일로 컴파일
- clean 명령어 정의

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

all: $(TARGET)

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

.c.o:
    $(CC) -c -o $@ $<

clean:
    rm *.o a.out
```

Makefile 읽기

- gcc 컴파일러 사용
- 최종 타겟 파일은 a.out
- 매크로 OBJS 정의
- 컴파일 옵션 -g
  (디버그 정보 표시)
- 타겟 파일을 만들기 위해
  OBJS를 사용하여, 아래 명령어 실행
- .c 소스코드를 .o 목적파일로 컴파일
- clean 명령어 정의

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

all: $(TARGET)

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

.c.o:
    $(CC) -c -o $@ $<

clean:
    rm *.o a.out
```

Makefile 읽기

- gcc 컴파일러 사용
- 최종 타겟 파일은 a.out
- 매크로 OBJS 정의
- 컴파일 옵션 -g
  (디버그 정보 표시)
- 타겟 파일을 만들기 위해
  OBJS를 사용하여, 아래 명령어 실행
- .c 소스코드를 .o 목적파일로 컴파일
- clean 명령어 정의

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

all: $(TARGET)

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

.c.o:
    $(CC) -c -o $@ $<

clean:
    rm *.o a.out
```

Makefile 읽기

- gcc 컴파일러 사용
- 최종 타겟 파일은 a.out
- 매크로 OBJS 정의
- 컴파일 옵션 -g
  (디버그 정보 표시)
- 타겟 파일을 만들기 위해
  OBJS를 사용하여, 아래 명령어 실행
- .c 소스코드를 .o 목적파일로 컴파일
- clean 명령어 정의

```
CC = gcc
TARGET = a.out
OBJS = main.o foo.o bar.o
CFLAGS = -g

all: $(TARGET)

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

.c.o:
    $(CC) -c -o $@ $<

clean:
    rm *.o a.out
```

감사합니다.