

리눅스 편집기와 **C** 프로그래밍

- gcc
 - gcc 기본 개념 및 사용법
 - gcc 옵션

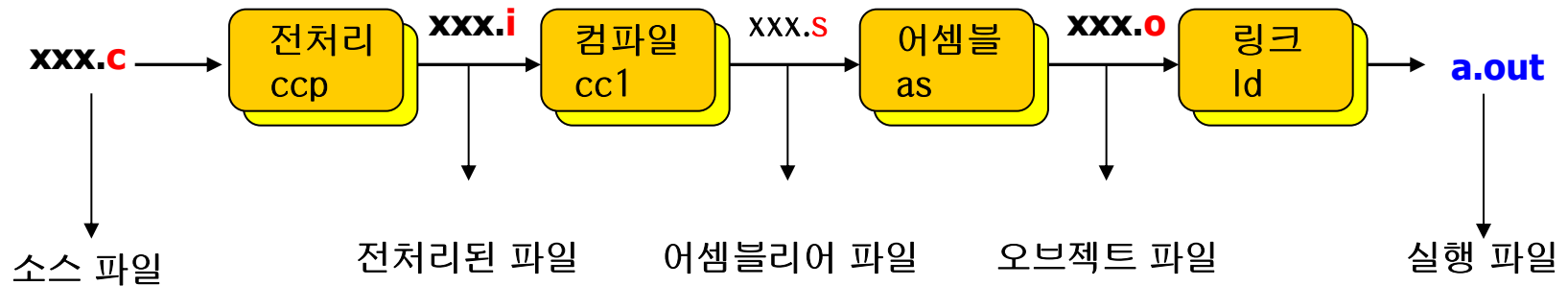
□ gcc란 ?

- 원래는 “GNU C Compiler”를 의미
- 1999년부터 “GNU Compiler Collection”을 의미한다. 따라서 C 언어뿐 아니라 C++, 오브젝티브 C, 포트란, 자바 등의 컴파일러를 포함하는 포괄적 의미를 가짐

gcc 기본 개념 및 사용법 (cont'd)

□ gcc란 ? (cont' d)

- 일반적으로 gcc를 컴파일러라고 하지만 정확히 말하면 gcc는 소스 파일을 이용해 실행 파일을 만들 때까지 필요한 프로그램을 차례로 실행시키는 툴



gcc 기본 개념 및 사용법 (cont'd)

□ gcc 기본 사용법

■ gcc는 파일 확장자에 따라 처리 방법을 달리 함

□ 예) 대표적인 확장자 .c 인 경우는 gcc로 전처리, 컴파일, 어셈블, 링크 과정을 거쳐야 실행 파일이 완성 됨

확장자	종류	처리 방법
.c	C 소스 파일	gcc로 전처리, 컴파일, 어셈블, 링크
.s	어셈블리어로 된 파일	어셈블, 링크
.S	어셈블리어로 된 파일	전처리, 어셈블, 링크
.o	오브젝트 파일	링크
.a .so	컴파일된 라이브러리 파일	링크

gcc 기본 개념 및 사용법 (cont'd)

□ 실습예제

[실습예제] a.c

```
#include <stdio.h>
int main()
{
    printf ("Hello Linux\n");

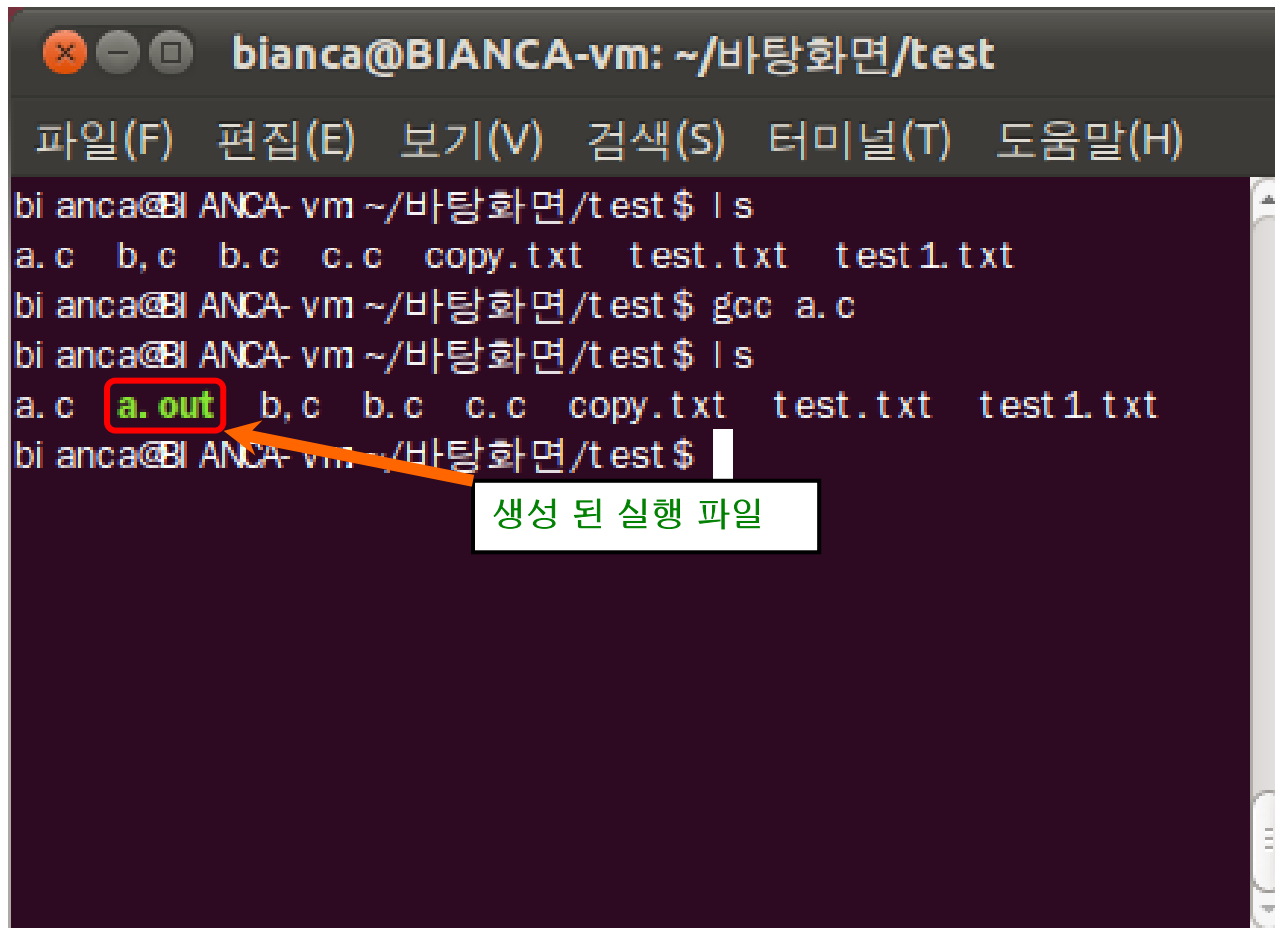
    return 0;
}
```

```
[root@localhost ~]$ gcc a.c
```

```
[root@localhost ~]$ ./a.out
```

gcc 기본 개념 및 사용법 (cont'd)

□ 실습예제



```
bianca@BIANCA-vm: ~/바탕화면/test
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c b.c b.c c.c copy.txt test.txt test1.txt
bianca@BIANCA-vm ~/바탕화면/test$ gcc a.c
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c a.out b.c b.c c.c copy.txt test.txt test1.txt
bianca@BIANCA-vm ~/바탕화면/test$
```

생성 된 실행 파일

□ gcc 옵션

옵 션	의 미
-c	소스파일을 오브젝트 파일로만 컴파일하고 링크하는 과정을 생략
-o	바이너리 형식의 출력 파일 이름을 지정하는데, 지정하지 않을 시 a.out 이라고 기본이름 생성
-I	헤더 파일을 검색하는 디렉토리 목록을 추가
-S	소스파일을 어셈블리 파일로만 컴파일
-E	소스파일을 전처리 단계까지만 처리
-L	라이브러리 파일을 검색하는 디렉토리 목록을 추가
-l	라이브러리 파일을 컴파일 시 링크
-g	바이너리 파일에 표준 디버깅 정보를 포함
-ggdb	바이너리 파일에 GNU 디버거인 gdb 만이 이해할 수 있는 많은 디버깅 정보를 포함시킴
-O	컴파일 코드를 최적화 함
-Olevel	최적한 <i>level</i> 단계를 지정(level은 1~3이 존재)
-DFOO=RAR	명령라인에서 BAR 값을 가지는 FOO 라는 선행 처리기 매크로를 정의
-static	정적 라이브러리에 링크

gcc 옵션 (cont'd)

□ gcc 옵션 (cont'd)

옵 션	의 미
-ansi	표준과 충돌하는 GNU 확장안을 취소, ANSI/ISO C 표준을 지원, ANSI 호환코드를 보장 안 함
-traditional	과거 스타일의 함수 정의 형식과 같이 전통적인 K&R C언어 형식을 지원
-MM	make 호환의 의존성 목록을 출력
-V	컴파일의 각 단계에서 사용되는 명령을 보여 줌
-llib	Link시 해당 lib를 같이 link하게 함

gcc 옵션 (cont'd)

□ -o 옵션

- 생성되는 출력 파일 이름을 지정

■ Syntax

gcc -o 출력파일이름 소스파일이름

gcc 소스파일이름 -o 출력파일이름

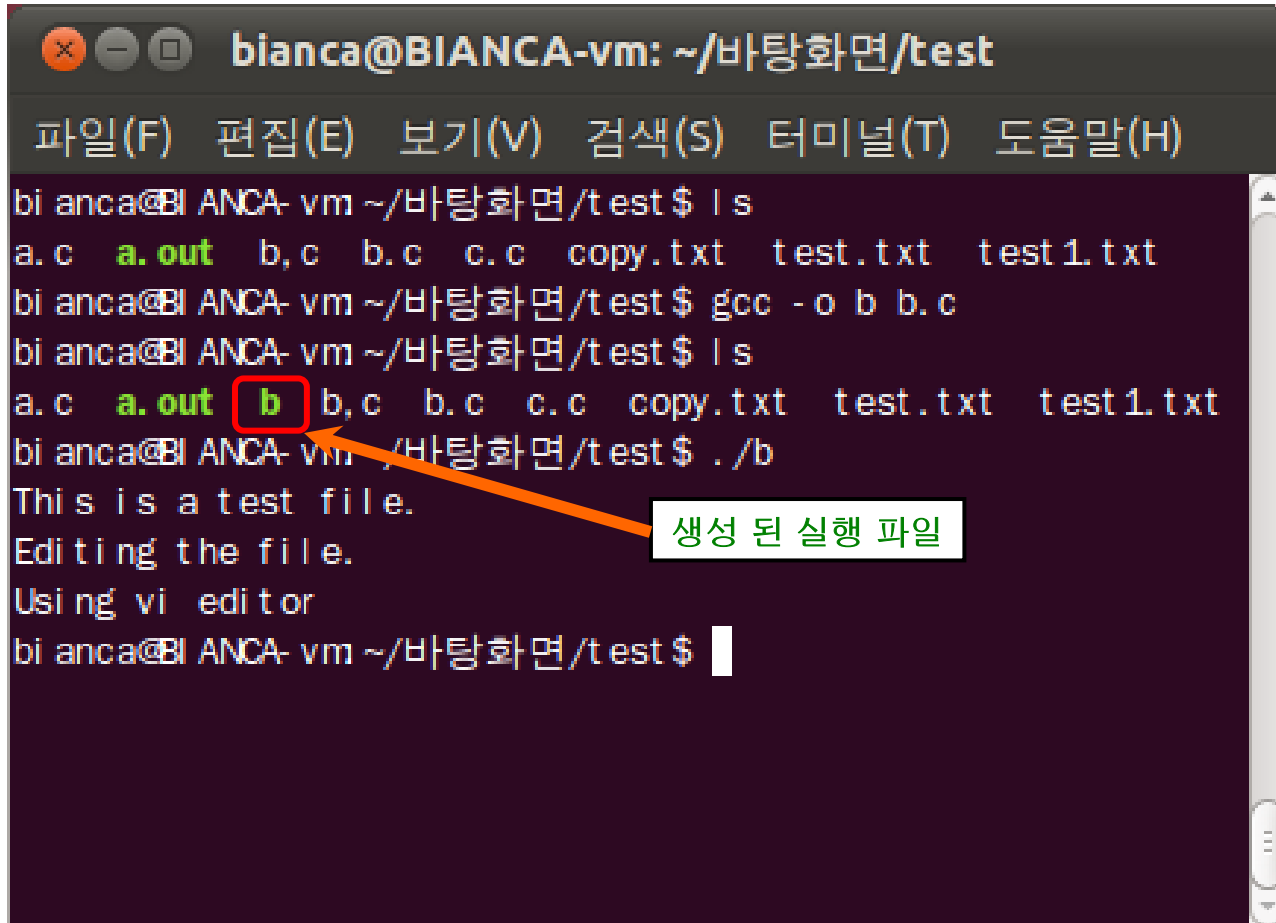
(출력파일과 소스파일의 순서는 바뀌어도 상관없음)

```
[root@localhost ~]$ gcc -o file file.c
```

- file.c 소스파일에 file 이라는 출력파일 이름을 지정해 주어서 a.out 이라는 기본 파일을 생성하지 않음

gcc 옵션 (cont'd)

□ -o 옵션(cont'd)



```
bianca@BIANCA-vm: ~/바탕화면/test
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c  a.out  b.c  c.c  copy.txt  test.txt  test1.txt
bianca@BIANCA-vm ~/바탕화면/test$ gcc -o b b.c
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c  a.out  b  b.c  c.c  copy.txt  test.txt  test1.txt
bianca@BIANCA-vm ~/바탕화면/test$ ./b
This is a test file.
Editing the file.
Using vi editor
bianca@BIANCA-vm ~/바탕화면/test$
```

생성된 실행 파일

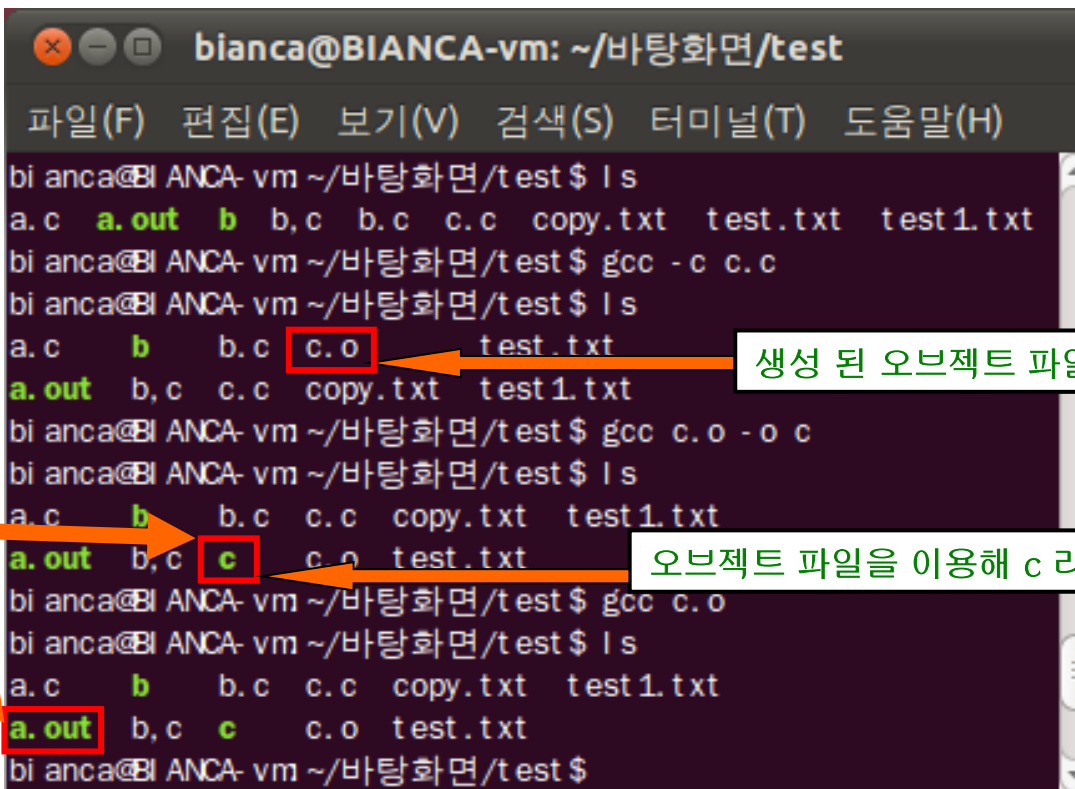
gcc 옵션 (cont'd)

□ -c 옵션

- 소스를 오브젝트 파일로만 컴파일하고 링크하는 과정을 생략

■ Syntax

gcc -c 소스파일이름



```
bianca@BIANCA-vm: ~/바탕화면/test
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c  a.out  b.c  b.c  c.c  copy.txt  test.txt  test1.txt
bianca@BIANCA-vm ~/바탕화면/test$ gcc -c c.c
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c  b.c  c.o  test.txt
a.out b.c c.c copy.txt test1.txt
bianca@BIANCA-vm ~/바탕화면/test$ gcc c.o -o c
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c  b.c  c.c  copy.txt  test1.txt
a.out b.c c  c.o  test.txt
bianca@BIANCA-vm ~/바탕화면/test$ gcc c.o
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c  b.c  c.c  copy.txt  test1.txt
a.out b.c c  c.o  test.txt
bianca@BIANCA-vm ~/바탕화면/test$
```

오브젝트 파일을 이용해 생성된 실행 파일

생성된 오브젝트 파일

오브젝트 파일을 이용해 c 라는 실행 파일 생성

gcc 옵션 (cont'd)

□ -c 옵션 (cont'd)

■ 분리 컴파일

- 여러 파일로 분리 작성된 하나의 프로그램을 컴파일

[예제 1] main.c

```
extern void hi() ;  
main()  
{  
    hi() ;  
}
```

[예제 2] hi.c

```
#include <stdio.h>  
void hi()  
{  
    printf ("Linux World \n");  
}
```

```
[root@localhost ~]$ gcc main.c hi.c -o test
```

```
[root@localhost ~]$ gcc -c main.c
```

```
[root@localhost ~]$ gcc -c hi.c
```

```
[root@localhost ~]$ gcc main.o hi.o -o test
```

gcc 옵션 (cont'd)

□ -c 옵션 (cont'd)

```
bianca@BIANCA-vm: ~/바탕화면/test
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c      b      b.c  c.c  copy.txt  test1.txt
a.out    b,c    c    c.o  test.txt
bianca@BIANCA-vm ~/바탕화면/test$ vi mai n.c
bianca@BIANCA-vm ~/바탕화면/test$ vi hi.c
bianca@BIANCA-vm ~/바탕화면/test$ gcc mai n.c hi.c -o test
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c      b      b.c  c.c  copy.txt  mai n.c  test1.txt
a.out    b,c    c    c.o  hi.c      test
bianca@BIANCA-vm ~/바탕화면/test$ ./test
Linux World
bianca@BIANCA-vm ~/바탕화면/test$
```

```
bianca@BIANCA-vm: ~/바탕화면/test
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c      b      b.c  c.c  copy.txt  mai n.c  test1.txt
a.out    b,c    c    c.o  hi.c      test.txt
bianca@BIANCA-vm ~/바탕화면/test$ gcc -c mai n.c
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c      b      b.c  c.c  copy.txt  mai n.c  test.txt
a.out    b,c    c    c.o  hi.c      mai n.o  test1.txt
bianca@BIANCA-vm ~/바탕화면/test$ gcc -c hi.c
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c      b      b.c  c.c  copy.txt  hi.o  mai n.o  test1.txt
a.out    b,c    c    c.o  hi.c      mai n.c  test.txt
bianca@BIANCA-vm ~/바탕화면/test$ gcc mai n.o hi.o -o test
bianca@BIANCA-vm ~/바탕화면/test$ ls
a.c      b      b.c  c.c  copy.txt  hi.o  mai n.o  test.txt
a.out    b,c    c    c.o  hi.c      mai n.c  test  test1.txt
bianca@BIANCA-vm ~/바탕화면/test$ ./test
Linux World
bianca@BIANCA-vm ~/바탕화면/test$
```

gcc 옵션 (cont'd)

□ -I 옵션

- 표준 디렉토리가 아닌 위치에 있는 헤더 파일의 디렉토리를 지정

■ Syntax

gcc 소스파일이름 -I 디렉토리이름

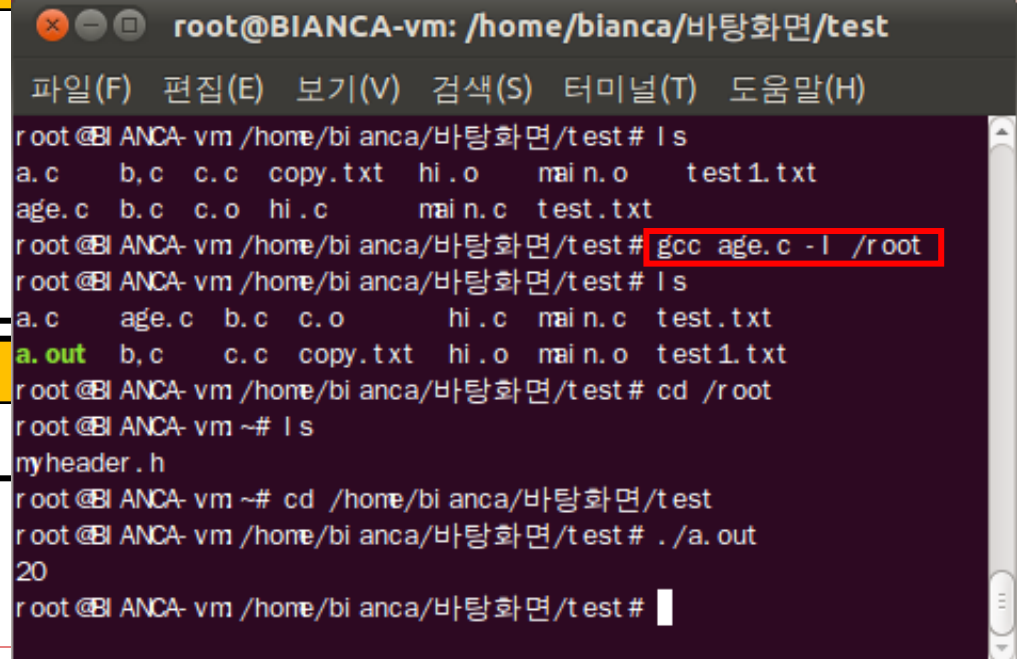
```
[root@localhost ~]$ gcc age.c -I 헤더파일이 있는 디렉토리 경로
```

[예제 1] age.c

```
#include <stdio.h>
#include "myheader.h"
main()
{
    printf("%d\n", AGE);
}
```

[예제 2] myheader.h

```
#define AGE 20
```



```
root@BIANCA-vm: /home/bianca/바탕화면/test
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
root@BIANCA-vm /home/bianca/바탕화면/test# ls
a.c  b.c  c.c  copy.txt  hi.o  main.o  test1.txt
age.c b.c  c.o  hi.c      main.c test.txt
root@BIANCA-vm /home/bianca/바탕화면/test# gcc age.c -I /root
root@BIANCA-vm /home/bianca/바탕화면/test# ls
a.c  age.c  b.c  c.o  hi.c  main.c  test.txt
a.out b.c  c.c  copy.txt  hi.o  main.o  test1.txt
root@BIANCA-vm /home/bianca/바탕화면/test# cd /root
root@BIANCA-vm ~# ls
myheader.h
root@BIANCA-vm ~# cd /home/bianca/바탕화면/test
root@BIANCA-vm /home/bianca/바탕화면/test# ./a.out
20
root@BIANCA-vm /home/bianca/바탕화면/test#
```

make 프로그램

Contents

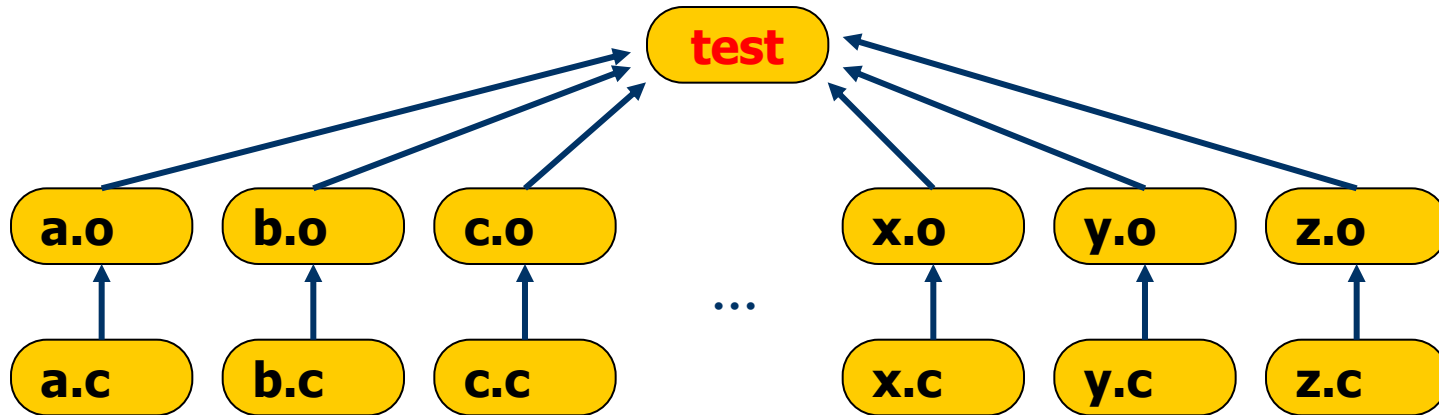
□ make 란?

□ Makefile

□ 몇 가지 문법 규칙

□ make 옵션

make란?



위와 같이 여러 파일로 구성된 프로그램이 있을 경우 c.c 소스파일을 수정하면, 모든 파일을 다시 컴파일하고 링크해야 수정이 반영된 test 파일이 생성됨.
즉, 파일을 하나만 수정해도 모든 파일을 다시 컴파일 해야함

make는 수정된 파일만 자동으로 알아내 컴파일 하고 수정하지 않은 파일에 대해서는 기존 오브젝트 파일을 그대로 이용하게 해주는 **유틸리티 툴**

□ Makefile

- Makefile은 애플리케이션의 구성방법을 **make**에 알려주는 텍스트 파일

□ 형식

대상(target) : 대상에 의존되는 파일1 [파일2 ...]

[*tab*간격] 명령(command)

예제)

대상

test :

test.c

의존부분(dependents) 또는 선결조건(prerequisites)

gcc test.c -o test

명령 : 대부분 컴파일 호출, 명령을 사용 시에는 반드시 **탭 문자**로 시작해야 함

□ Makefile 생성시 주의 사항

- 각 요소를 구분하는데 있어 콤마(,) 같은 건 사용하지 않고 공백으로 함
- 명령을 시작하기 전에는 항상 <TAB>을 넣음
 - 절대 스페이스 키나 다른 키는 사용해선 안됨
 - 그 밖의 다른 곳에서는 <TAB>을 사용하지 말 것
- Makefile 내에서 항목의 순서는 중요하지 않음
 - make는 어떤 파일이 어느 곳에 의존적인지 알아내어 올바른 순서로 명령을 수행

Makefile (cont'd)

□ Makefile 예제

[예제 1] test1.c

```
#include <stdio.h>
#include "a.h"
extern void func1();
extern void func2();
main()
{
    printf("test1\n");
    func1 ();
    func2 ();
}
```

[예제 2] test2.c

```
#include <stdio.h>
#include "a.h"
#include "b.h"
void func1()
{
    printf("test2 \n");
}
```

[예제 3] test3.c

```
#include <stdio.h>
#include "b.h"
#include "c.h"
void func2()
{
    printf("test3 \n");
}
```

Makefile (cont'd)

□ Makefile 예제 (cont'd)

① 헤더 파일 생성

```
[root@localhost ~]$ touch a.h
```

```
[root@localhost ~]$ touch b.h
```

```
[root@localhost ~]$ touch c.h
```

② Makefile 생성

[<Makefile]

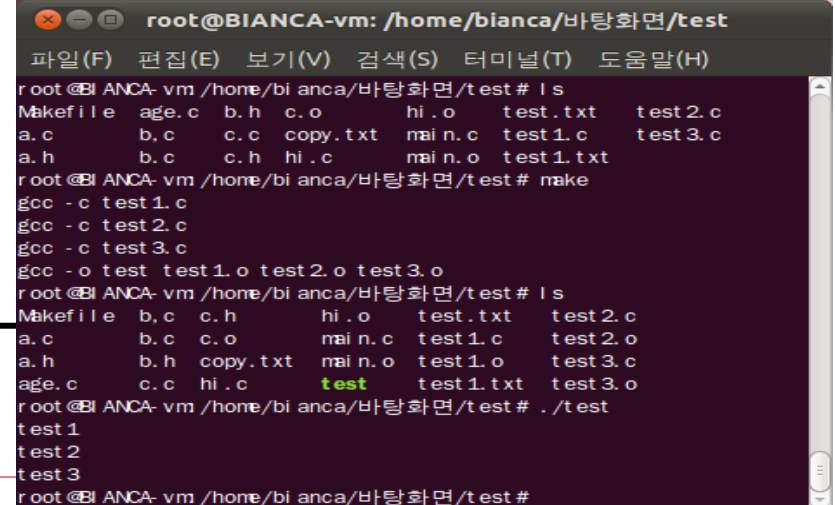
```
test: test1.o test2.o test3.o
    gcc -o test test1.o test2.o test3.o

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

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

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

③ make 실행 (#make)



```
root@BIANCA-vm: /home/bianca/바탕화면/test
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
root@BIANCA-vm: /home/bianca/바탕화면/test # ls
Makefile age.c b.h c.o hi.o test.txt test2.o
a.c b.c c.c copy.txt main.c test1.c test3.o
a.h b.c c.h hi.c main.o test1.txt
root@BIANCA-vm: /home/bianca/바탕화면/test # make
gcc -c test1.c
gcc -c test2.c
gcc -c test3.c
gcc -o test test1.o test2.o test3.o
root@BIANCA-vm: /home/bianca/바탕화면/test # ls
Makefile b.c c.h hi.o test.txt test2.o
a.c b.c c.o main.c test1.c test2.o
a.h b.h copy.txt main.o test1.o test3.c
age.c c.c hi.c test test1.txt test3.o
root@BIANCA-vm: /home/bianca/바탕화면/test # ./test
test1
test2
test3
root@BIANCA-vm: /home/bianca/바탕화면/test #
```

Makefile (cont'd)

□ Makefile 예제 (cont'd)

□ make clean

: 소스파일만 남기고 생성된 다른 파일들은 지워줌

[<Makefile]

```
test: test1.o test2.o test3.o
    gcc -o test test1.o test2.o test3.o
test1.o: test1.c a.h
    gcc -c test1.c
test2.o: test2.c a.h b.h
    gcc -c test2.c
test3.o: test3.c b.h c.h
    gcc -c test3.c

clean:
    rm -f test test1.o test2.o test3.o
```

□ 실행

□ #make clean

몇 가지 문법 규칙

□ 매크로(Macro)

- Makefile을 작성하다 같은 파일 이름을 여러 번 써야 하는 경우, 매크로를 사용하면 편리하고 명령을 단순화 시킬 수 있음

- “Macro makes makefile happy.”

```
M_NAME = value
```

- 매크로 사용시 대소문자 모두 가능
 - 보통 대문자로 쓰는 것이 관례
- Makefile 상단에 정의

몇 가지 문법 규칙 (cont'd)

□ 매크로 사용 예제

[Makefile]

OBJF = test1.o test2.o test3.o

매크로 정의

test: **\$(OBJF)**

gcc -o test **\$(OBJF)**

\$(OBJF) = test1.o test2.o test3.o

test1.o: test1.c a.h

gcc -c test1.c

test2.o: test2.c a.h b.h

gcc -c test2.c

test3.o: test3.c b.h c.h

gcc -c test3.c

clean:

rm -f \$(OBJF)

몇 가지 문법 규칙 (cont'd)

□ 내부 매크로

내부 매크로	의 미
\$@	현재 목표 파일의 이름
\$*	확장자를 제외한 현재 목표 파일의 이름
\$<	현재 필수 조건 파일 중 첫 번째 파일 이름
\$?	현재 대상보다 최근에 변경된 필수 조건 파일 이름
\$^	현재 모든 필수 조건 파일들

몇 가지 문법 규칙 (cont'd)

□ 내부 매크로 사용 예제

[Makefile]

```
OBJF = test1.o test2.o test3.o
```

```
test: $(OBJF)
```

```
gcc -o $@ $^
```

```
test1.o: test1.c a.h
```

```
gcc -c $<
```

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

```
gcc -c $*.c
```

```
test3.o: test3.c b.h c.h
```

```
gcc -c $*.c
```

```
clean:
```

```
rm -f $(OBJF)
```

현재 대상 파일의 이름을 의미하므로 test를 나타냄

OBJF로 정의된 매크로 값

현재 대상 파일 test1.o가 의존하는 필수 조건 파일 중 첫 번째 파일 이름을 의미 test1.c

확장자 .c 를 제외한 현재 대상 파일의 이름을 의미
각 각 test2 , test3 을 의미

몇 가지 문법 규칙 (cont'd)

□ 접미사 규칙

예제)

```
test: test1.o test2.o
      gcc -o test test1.o test2.o
.c.o:                                     ...①
      gcc -c $< $(CFLAGS)                ...②
```

① .c.o

.c 라는 확장자를 가진 파일을 사용해 .o 라는 확장자를 가진 파일을 만들 것임을 make에 알리는 역할을 한다.

② \$<

확장자가 .c인 파일명을 의미한다.

\$(CFLAGS) C 컴파일러를 위한 플래그를 위해 미리 정의된 변수

몇 가지 문법 규칙 (cont'd)

□ 패턴 규칙

- 암시적 규칙에 의존했을 경우 일어나는 오류방지
- 접미사 규칙과 비슷하나 더 뛰어난 기능을 가짐

[Makefile]

```
OBJF = test1_d.o test2_d.o test3_d.o
```

```
test: $(OBJF)
```

```
    gcc -o $@ $(OBJF)
```

```
%_d.o: %.c
```

```
    gcc -c -g $< -o $@
```

```
clean:
```

```
    rm -f $(OBJF)
```

확장자 .c 인 모든 파일에 대해 _d 를 붙인 오브젝트 파일을 생성하겠다는 의미

-g 옵션을 주어 컴파일 시 디버깅 정보를 삽입

make 옵션

옵션	의미
-f 파일이름	GNUmakefile, makefile, Makefile 외의 이름을 갖는 make 파일을 실행시킬 때 사용자 임의의 파일 이름을 지정한다.
-n	make가 실행하는 명령을 출력만 하고 실제로 실행 하지 않는다
-W 파일이름	파일 이름이 변경된 것처럼 동작한다.
-s	make가 실행하는 명령을 출력하지 않고 실행한다.
-r	make의 모든 내장 규칙을 사용할 수 없다.
-d	make 실행 시 많은 디버깅 정보도 같이 출력한다.
-k	한 대상을 구성하는데 실패해도 다음 대상을 계속 구성한다.

make 옵션 (cont'd)

□ -f 옵션

기 능 : 표준 파일 외의 파일을 실행

기본형 : `make -f 파일이름`

□ -n , -W 옵션

기 능 : 실행하지는 않으면서 실행해야 할 명령을 출력한다.

기본형 : `make -n`

기 능 : 파일이 변경 된 것처럼 동작한다.

기본형 : `make -W파일이름`

-n 은 `make: 'test' is up to date.` 라는 메시지만 출력하고 아무런 명령도 출력하지 않는다.
그러므로 -W 와 같이 사용해 특정 파일이 변경 된 것처럼 동작하게 한다.
-n 과 같이 사용되어 대상 파일에는 영향을 주지 않고, 명령 수행을 한다.

make 옵션 (cont'd)

□ -s 옵션

기 능 : 실행하는 명령을 출력하지 않고 실행한다.

기본형 : `make -s 파일이름`

-s 는 실행 명령이 많고 출력이 불필요한 경우에 유용하다.

□ -r 옵션

기 능 : 모든 내장 규칙을 사용할 수 없게 한다..

기본형 : `make -r`

make 옵션 (cont'd)

□ -d 옵션

기 능 : 디버깅 정보를 출력한다.

기본형 : `make -d`

-d 는 gcc의 -g 옵션과 마찬가지로 make 파일도 make 실행 시 디버깅 정보를 출력

□ -k 옵션

기 능 : 명령에 실패해도 계속 동작한다.

기본형 : `make -k`

-k 는 make가 명령을 실행하는 데 실패해도 잘못된 명령을 실행 했다는 메시지만 출력될 뿐 나머지 명령은 계속 실행한다.

- [1] 이종우, 류연승, “**LINUX 관리자 가이드 3/e**”, 사이텍미디어, pp. 131.
- [2] 박승규, “**RedHat Linux 9**”, 한빛미디어, pp. 229 - 245.
- [3] 이만용, “**러닝 리눅스**”, 한빛미디어, pp. 357 - 385.
- [4] 한국정보통신인력개발센터, “**리눅스마스터 2급 표준 교재**”, 사이텍미디어, pp. 211 - 229.