리눅스 C 프로그래밍

2007. 11. 14

안 효 창

C언어란?

- 1972년, 데니스 리치
- 뛰어난 기능과 융통성을 제공
- 대부분의 시스템 소프트웨어를 구현하는 언어
- 고급 언어면서 저급 언어처럼 비트나 바이트 처리, 그리고 포인터에 의한 주소 처리

리눅스에서의 C언어

- 리눅스의 대부분이 C언어로 작성
- 리눅스 환경의 대부분의 애플리케이션을 C언어로 작성
- 결국 리눅스 환경에서 프로그래밍을 공부하거나 개발하려는 분들은 리눅스와 영원히 함께 할 C언어를 이용하는 것이 바람직하다.

프로그래밍 툴 관련 사이트

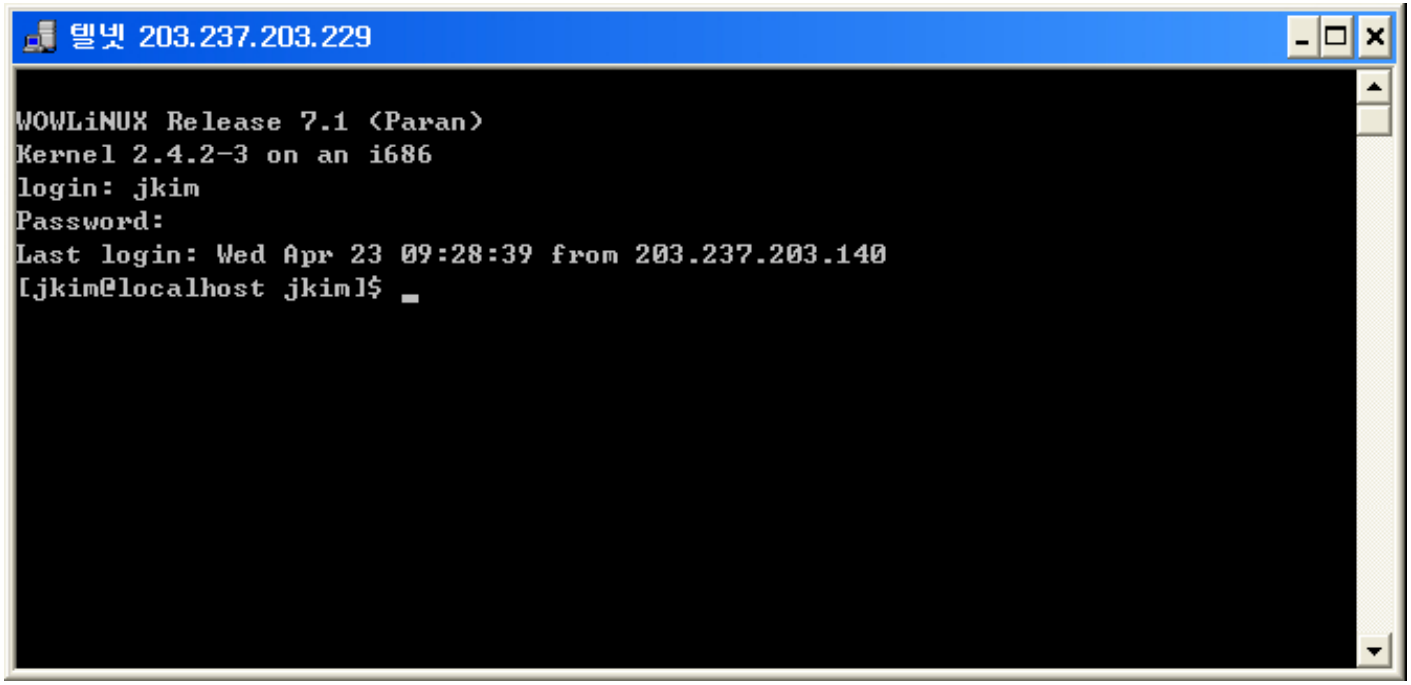
http://gcc.gnu.org	gcc 매뉴얼과 프로그램을 제공한다.
http://www.gnu.org/software/make	make 매뉴얼과 프로그램을 제공한다.
http://www.gnu.org/software/gdb	gdb 매뉴얼과 프로그램을 제공한다.

리눅스/프로그래밍 관련 사이트

http://www.tldp.org	리눅스와 관련된 여러 가지 기술 자료들을 제공한다.
http://www.kldp.org	리눅스와 관련된 여러 가지 기술 자료들을 한글판으로 제공한다.
http://www.unix.or.kr	유닉스/리눅스 프로그래밍에 대한 다양한 자료를 제공한다.
http://www.lug.or.kr	리눅스 팁을 제공하고 질답 코너를 운영한다.
http://cafe.naver.com/computerbook	[Linux & Unix C] 코너에서 리눅스 C 프로그래밍에 대한 질답 코너를 운영하고 관련 소식을 제공한다.

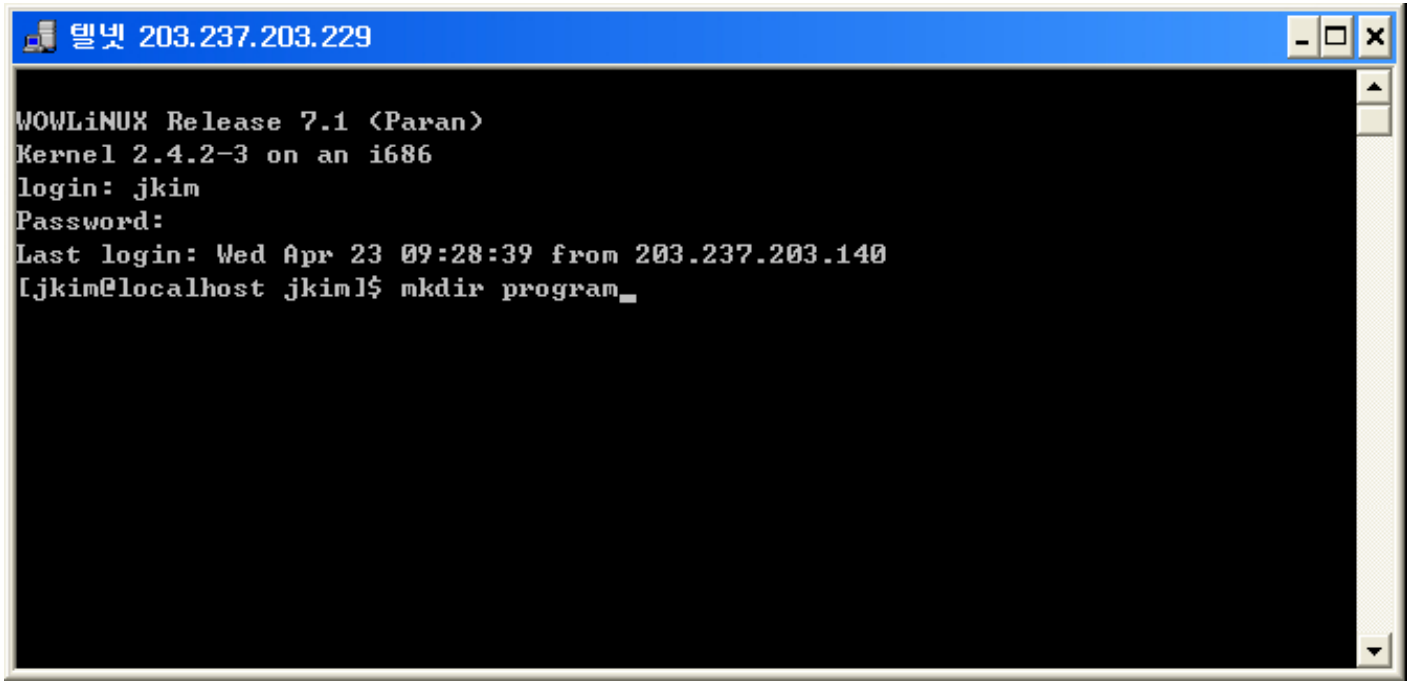
리눅스에서 C 프로그램 실습하기

- 리눅스 시스템 로그인하기



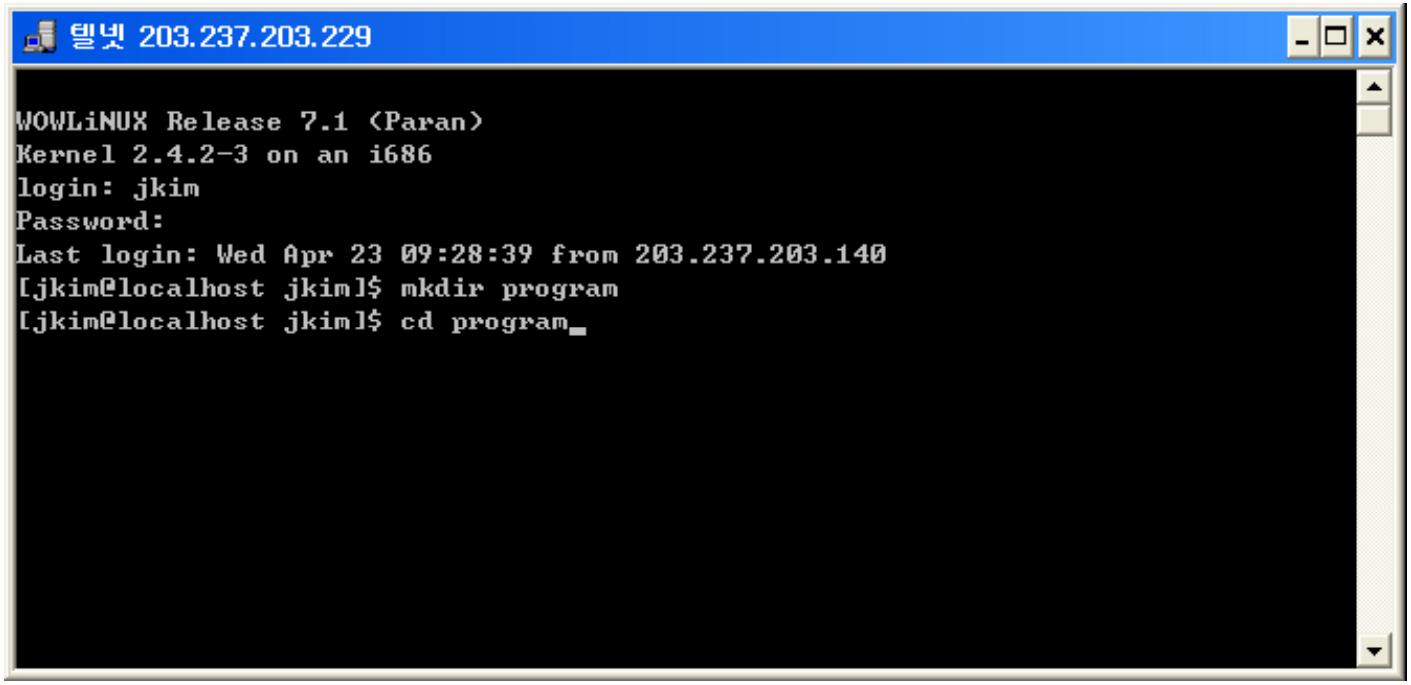
리눅스에서 C 프로그램 실습하기

- 디렉토리 만들기



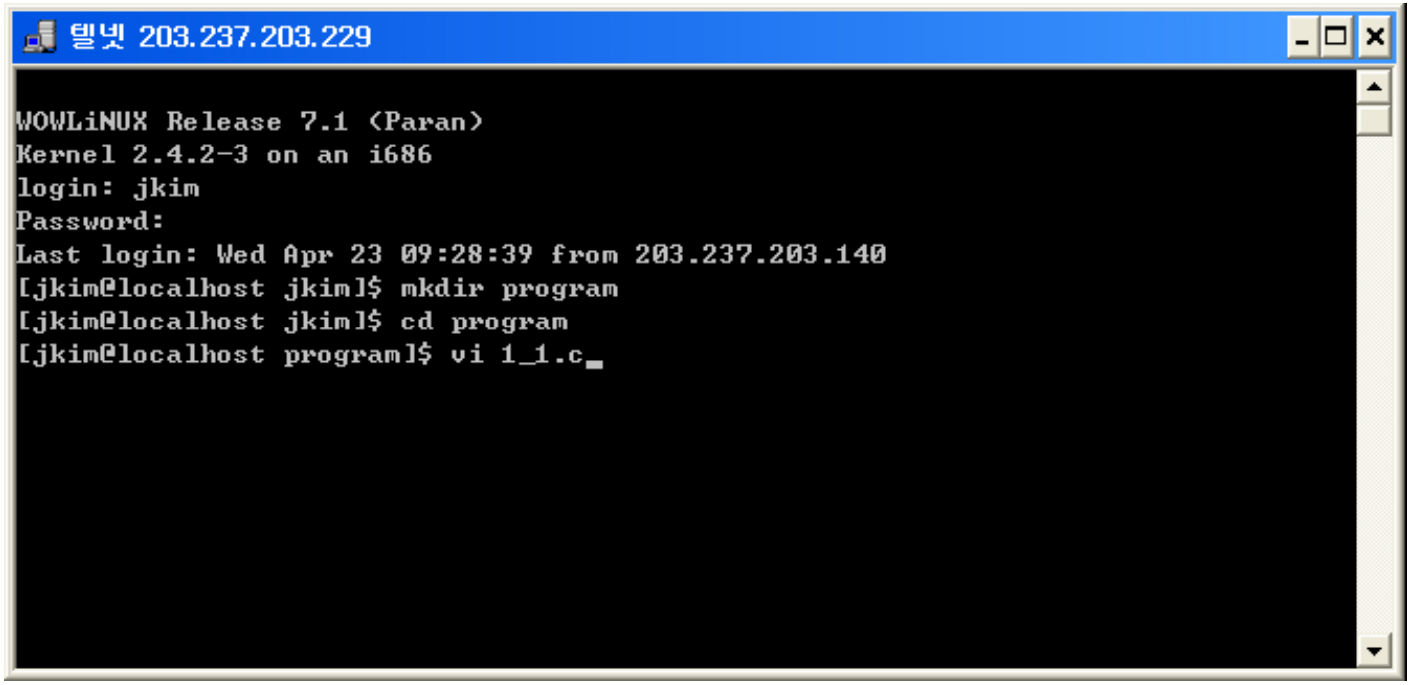
리눅스에서 C 프로그램 실습하기

- 디렉토리 이동하기



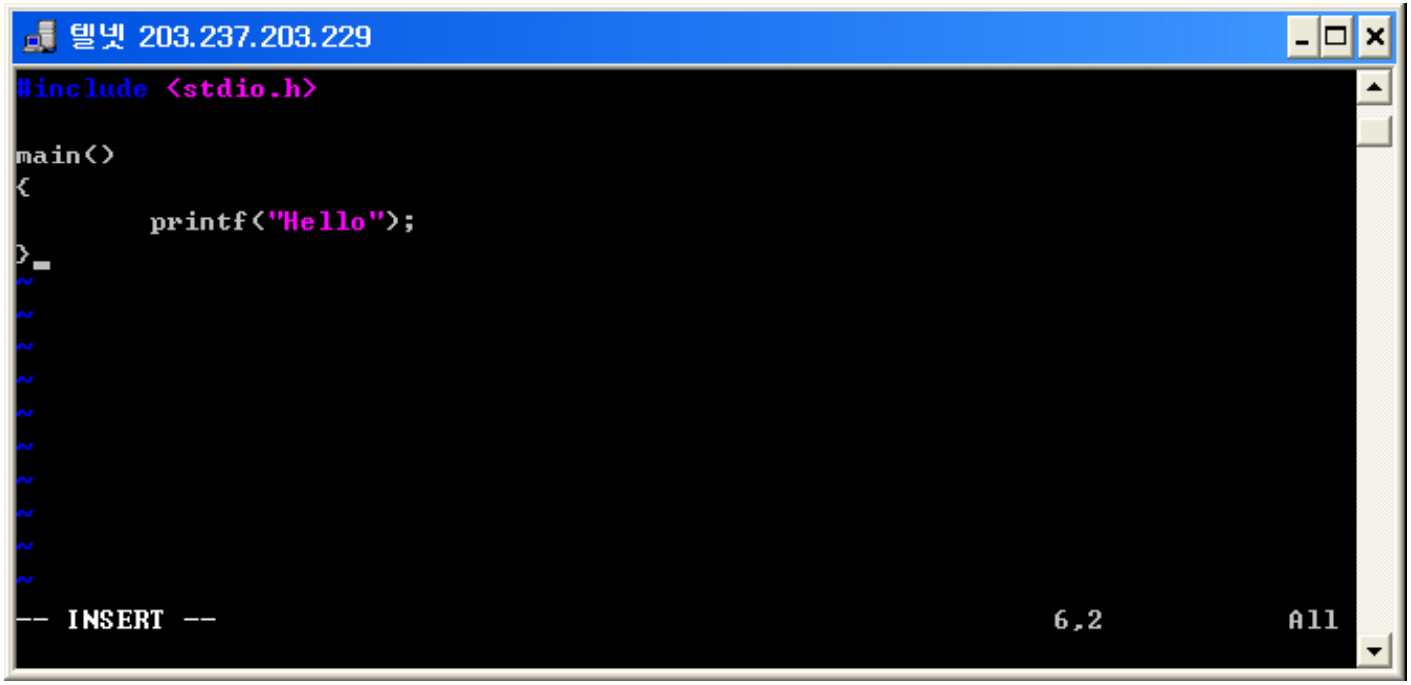
리눅스에서 C 프로그램 실습하기

- vi 실행하기



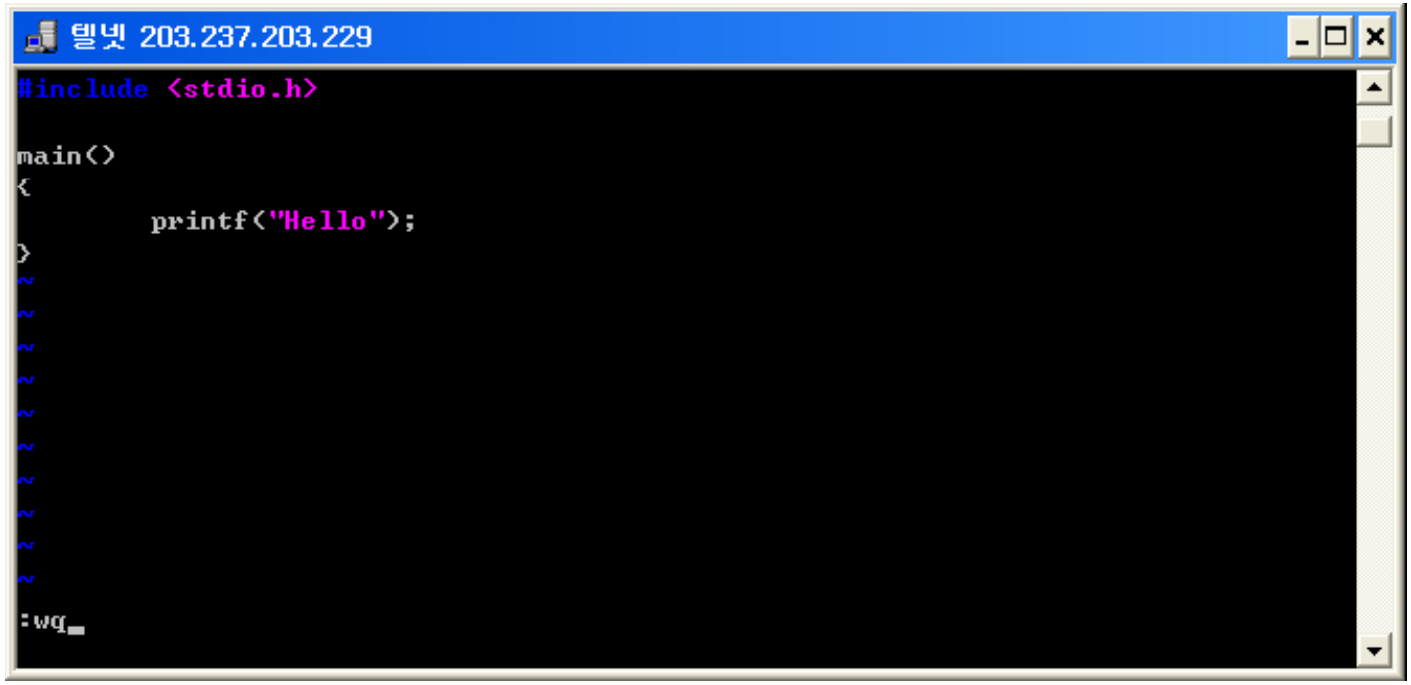
리눅스에서 C 프로그램 실습하기

- <i>키 누르고 편집하기



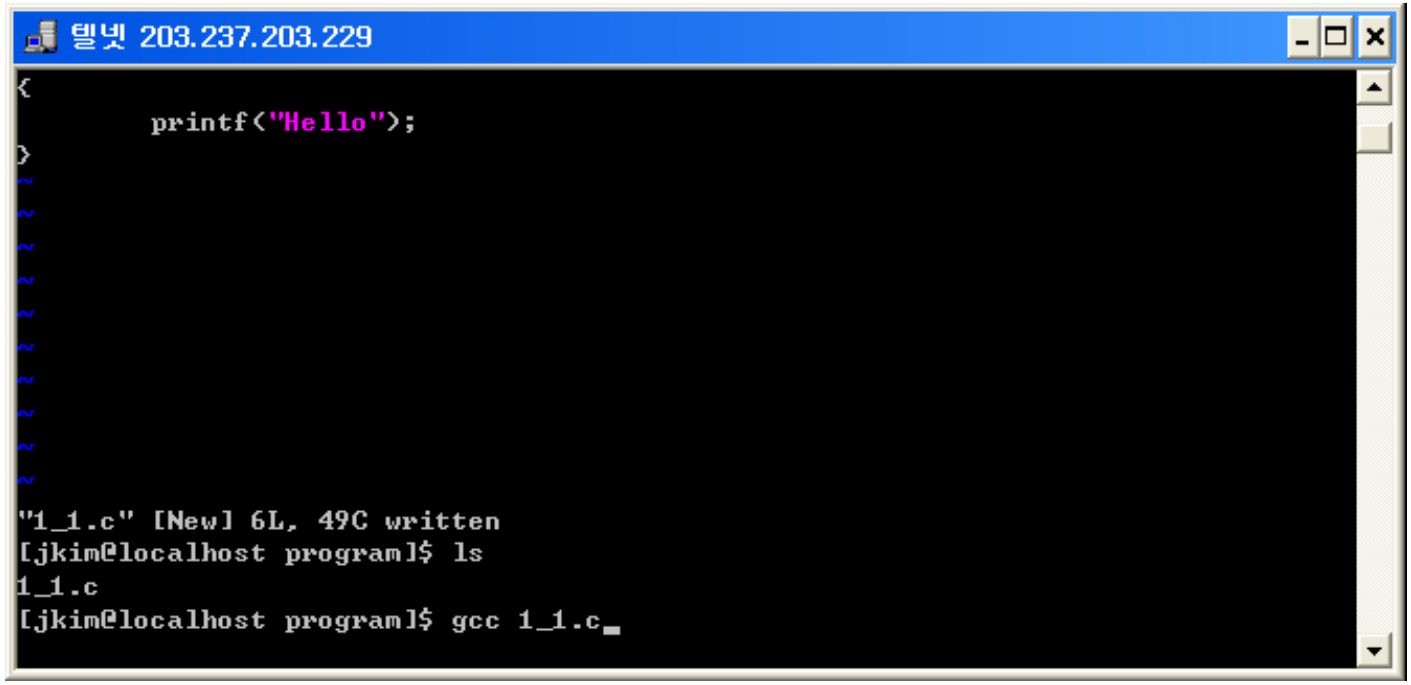
리눅스에서 C 프로그램 실습하기

- <Esc>키, 콜론(:), <w>,<q>키 누르고 종료하기



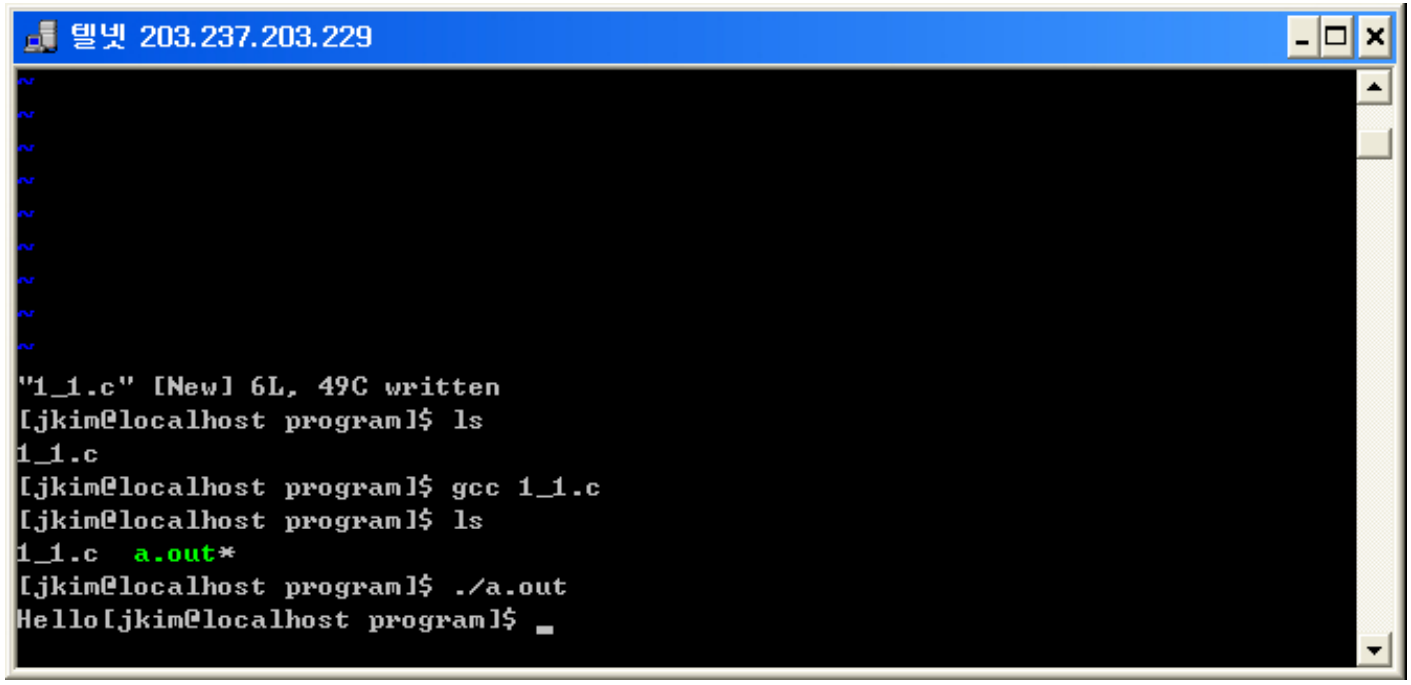
리눅스에서 C 프로그램 실습하기

- gcc로 컴파일하기

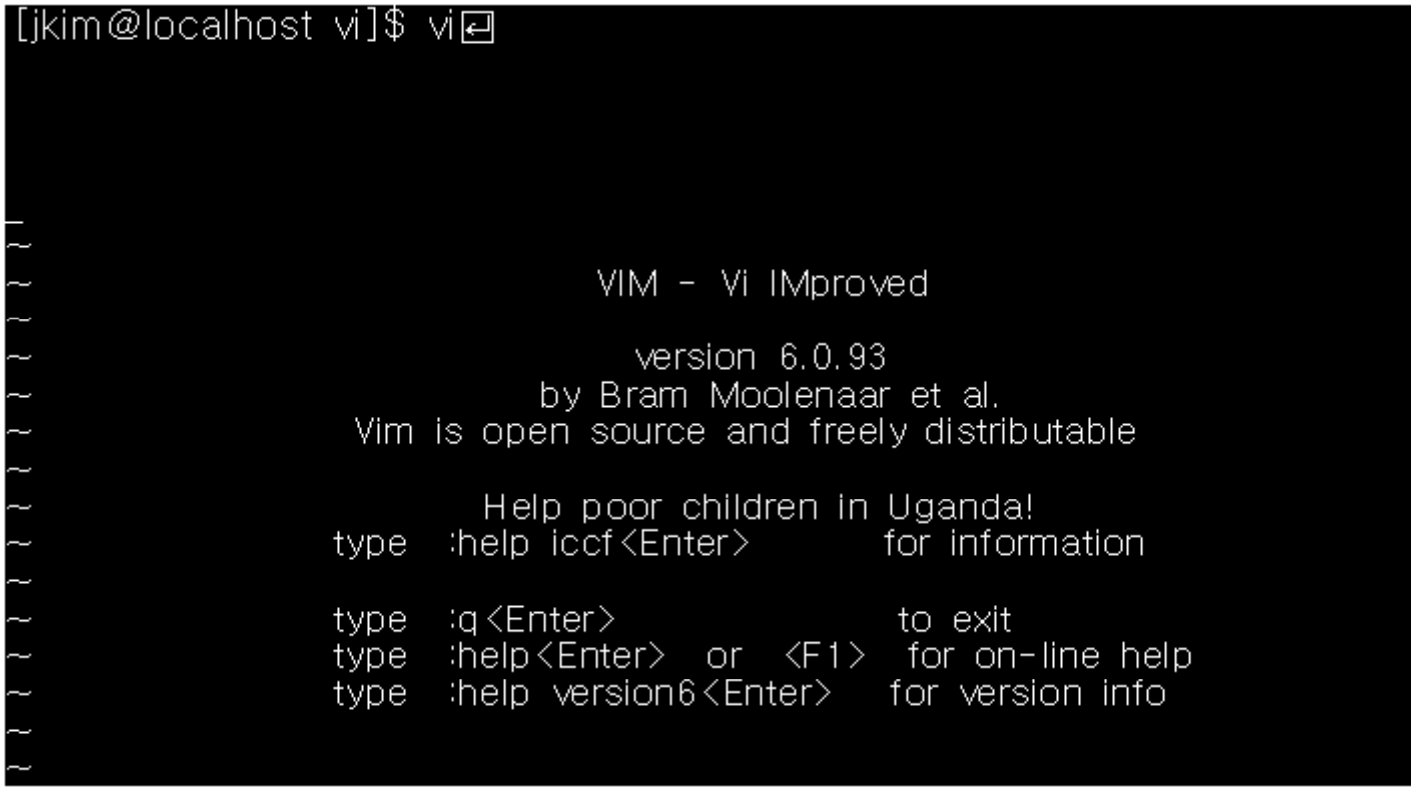


리눅스에서 C 프로그램 실습하기

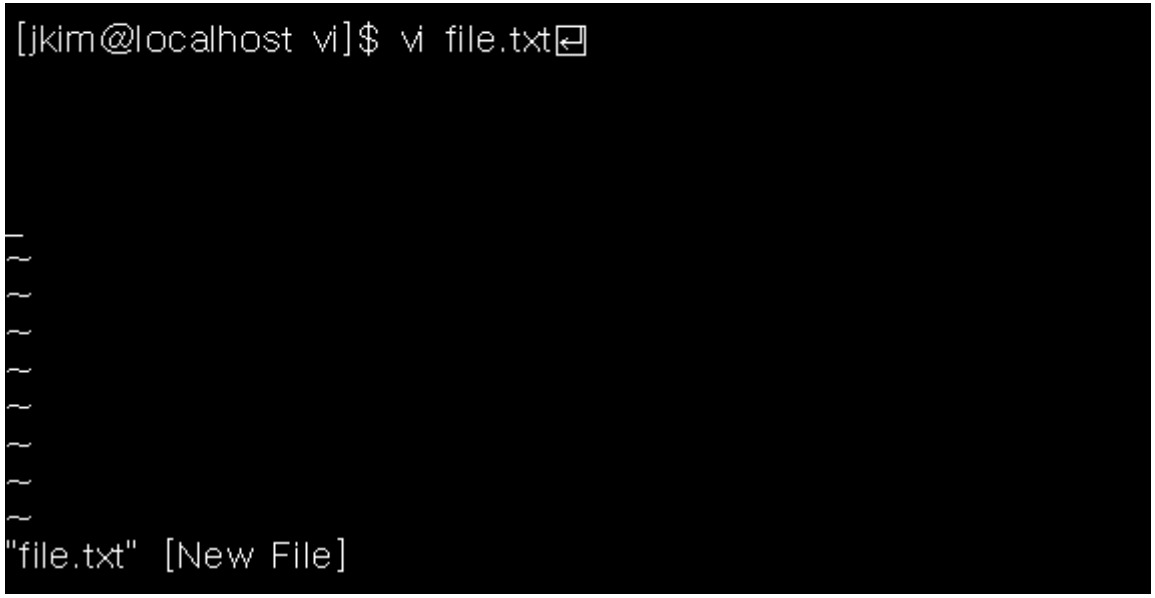
- 실행시키기



vi 실행하기

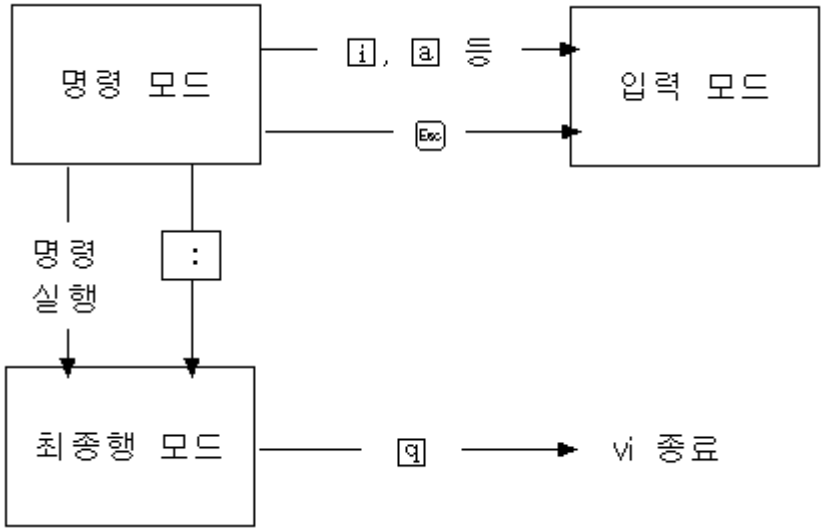

[jkim@localhost vi]$ vi
VIM - Vi IMproved
version 6.0.93
by Bram Moolenaar et al.
Vim is open source and freely distributable
Help poor children in Uganda!
type :help iccf<Enter> for information
type :q<Enter> to exit
type :help<Enter> or <F1> for on-line help
type :help version6<Enter> for version info

vi 실행하기



vi의 세 가지 모드

모드	의미	상태
입력 모드	글을 입력하는 모드	i, I, a, A, o, O 등을 누른 상태
명령 모드	위치 이동과 편집 기능을 사용하는 모드	Esc 키를 누른 상태
최종행 모드	파일 열기, 저장, 문자열 검색, 치환 등의 기능을 사용하는 모드	Esc 키를 누르고, 콜론(:)을 누른 상태



입력 모드


[jkim@localhost vi]$ vi file.txt
~
~
~
~
~
~
~
~
-- INSERT --


Hello c programming
Linux programming
~
~
~
~
~
~
~
-- INSERT --

입력 모드

키 입력	의미
i	Insert, 현재 커서의 위치에 글자를 삽입
I	Insert, 커서가 있는 줄(line)의 맨 앞에 글자를 삽입
a	Append, 현재 커서 위치의 다음 칸에 글자를 추가
A	Append, 커서가 있는 줄(line)의 맨 뒤에 글자를 추가
o	Open line, 현재의 줄 다음에 새로운 줄을 삽입
O	Open line, 현재의 줄 앞에 새로운 줄을 삽입

명령 모드


Hello c programming
Linux programming
~
~


Hello c programming
Linux programming
~
~


Hello c programming
Linux rogramming
~
~


Hello c programming
Linux Programming
~
~

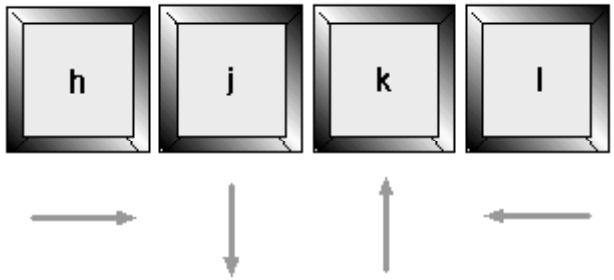
최종행 모드


Hello c programming
Linux Programming
~
~
:_


Hello c programming
Linux Programming
~
~
:wq


~
~
"file.txt" 2L, 38C written
[jkim@localhost vi]$

상하좌우로 이동하기



기타 이동키

키	의미	예
0(숫자)	현재 행의 맨 앞으로 이동한다.	3행의 a로 이동한다.
$	현재 행의 맨 마지막 문자로 이동한다.	3행의 n으로 이동한다.
^	현재 행의 맨 앞 문자로 이동한다.	3행의 a로 이동한다.
gg	첫 행의 맨 앞으로 이동한다.	1행의 a로 이동한다.
G	마지막 행으로 이동한다.	9행의 a로 이동한다.
nG	n 행으로 이동한다.	7G를 누르면 7행의 a로 이동한다.
H	스크린의 첫 행의 맨 앞으로 이동한다.	1행의 a로 이동한다.
M	스크린의 가운데 행의 맨 앞으로 이동한다.	5행의 a로 이동한다.
L	스크린의 맨 마지막 행의 맨 앞으로 이동한다.	9행의 a로 이동한다.
{	단락의 맨 앞으로 이동한다.	1행의 a로 이동한다.
}	다음 단락의 선두로 이동한다.	4행의 맨 앞으로 이동한다.
↵	다음 행의 맨 앞으로 이동한다.	4행의 맨 앞으로 이동한다.

지우기

x 키	현재 위치의 문자가 삭제된다.
X 키	현재 위치에서 왼쪽에 있는 문자가 삭제된다.
dd 키	현재 행을 삭제한다.
dw 키	커서 위치에서 단어 끝까지 삭제한다.
D 키	커서 위치에서 행 끝까지 삭제한다.

기타

- <y> 키 : 복사
- <d> 키 : 오려두기
- <p> 키, <P>키 : 붙이기
- <y><y> 키 : 현재 행 복사하기
- <d><d> 키 : 현재 행 오려두기

수정모드

- 명령 모드에서 <R>키

검색하기

- /문자열 : 아래쪽으로 검색하기
- ?문자열 : 위쪽으로 검색하기

문자열 치환하기

```
:%s/바꾸고자하는문자열/바꿀문자열/g
```

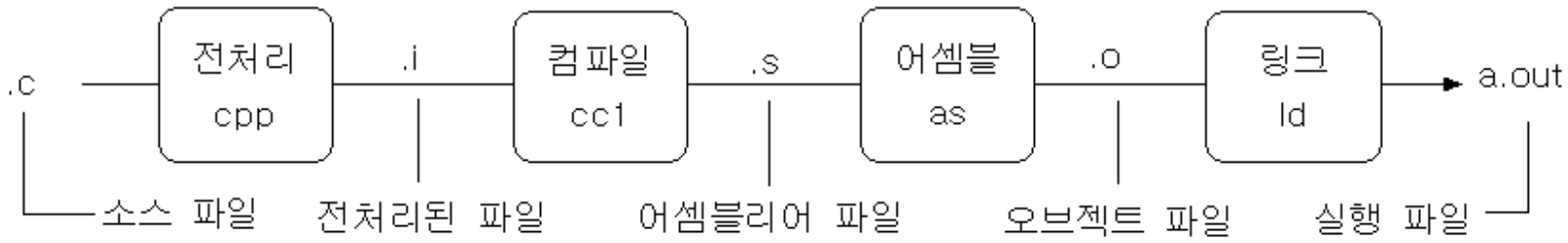
```
:%s/hab/sum/g
```

표현	의미
:%s/cheju/jeju/g	파일 내에 있는 모든 문자열 cheju를 jeju로 치환한다.
:%s/cheju/jeju/10	파일 내에 있는 문자열 cheju를 처음부터 10번째 까지만 jeju로 치환한다.
:s/cheju/jeju/	현재 커서가 위치해 있는 행의 첫 번째 문자열 cheju를 jeju로 치환한다.
:s/cheju/jeju/g	현재 커서가 위치해 있는 행의 모든 문자열 cheju를 jeju로 치환한다.
:10,20s/cheju/jeju/g	10행부터 20행까지의 문자열 cheju를 jeju로 치환한다.
:.,$s/cheju/jeju/g	현재 행(.)부터 마지막 행($)까지의 문자열 cheju를 jeju로 치환한다.

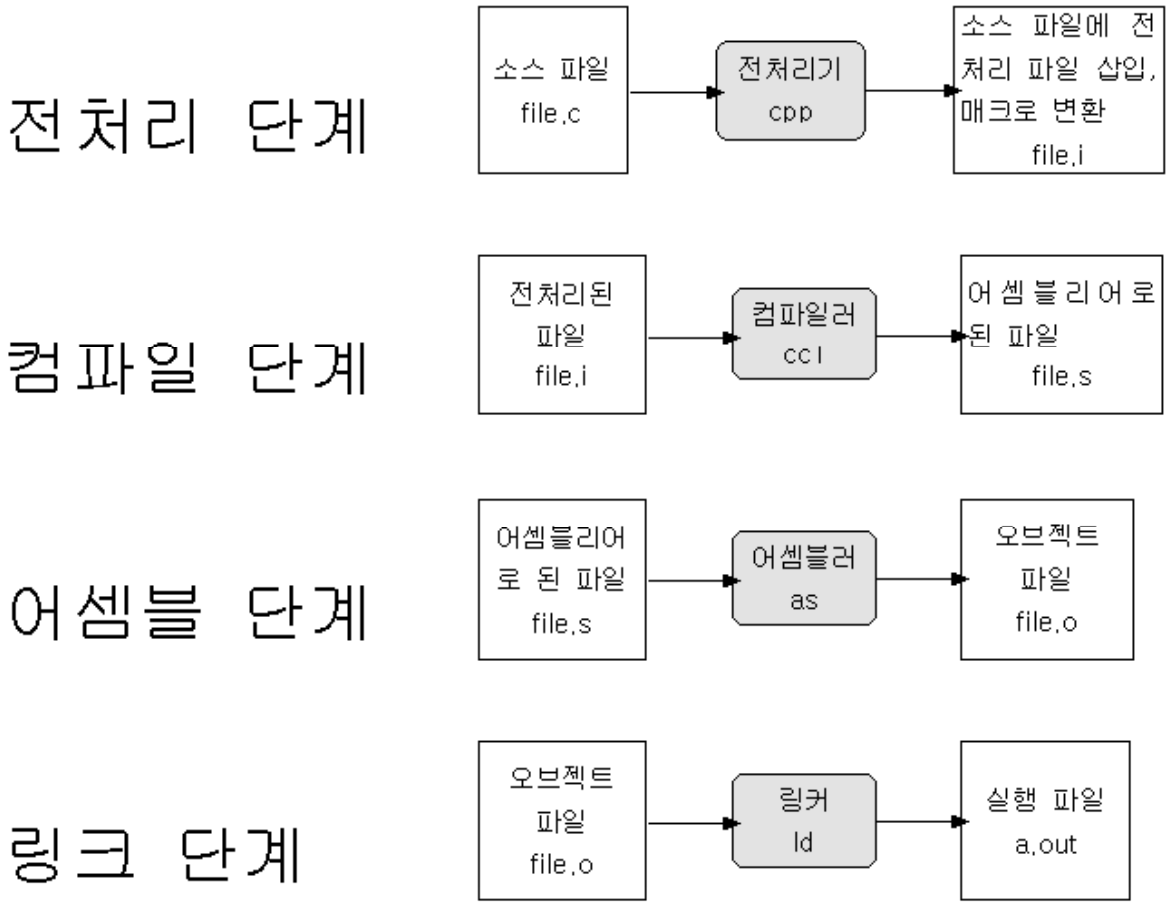
파일 다루기와 종료하기

키	의미
:e 파일이름	지정한 파일을 연다.
:w 파일이름	지정한 파일 이름으로 저장한다.
:w	저장한다.
:q	vi를 종료한다.
:q!	무조건 vi를 종료한다.
:wq	수정된 내용 저장하고 vi를 종료한다.
ZZ	수정된 내용 저장하고 vi를 종료한다.

gcc 동작 과정


.c
전처리
cpp
.i
컴파일
cc1
.s
어셈블
as
.o
링크
ld
a.out
소스 파일
전처리된 파일
어셈블리어 파일
오브젝트 파일
실행 파일

gcc 각 단계


전처리 단계
소스 파일
file.c
전처리기
cpp
소스 파일에 전처리 파일 삽입, 매크로 변환
file.i
컴파일 단계
전처리된 파일
file.i
컴파일러
cc1
어셈블리어로 된 파일
file.s
어셈블리 단계
어셈블리어로 된 파일
file.s
어셈블러
as
오브젝트 파일
file.o
링크 단계
오브젝트 파일
file.o
링커
ld
실행 파일
a.out

파일 확장자에 따른 처리 방법

확장자	종류	처리 방법
.c	C 소스 파일	gcc로 전처리, 컴파일, 어셈블, 링크
.C .CC	C++ 소스 파일	g++로 전처리, 컴파일, 어셈블, 링크
.i	전처리된 C 파일	gcc로 컴파일, 어셈블, 링크
.ii	전처리된 C++ 파일	g++로 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 실행하기

file.c 프로그램 작성 후

```
[jkim@localhost chap16]$ gcc file.c↵
[jkim@localhost chap16]$
```

```
$ gcc 소스파일이름
```

```
[jkim@localhost chap16]$ a.out↵
bash: a.out: command not found
[jkim@localhost chap16]$
```

```
[jkim@localhost chap16]$ ./a.out↵
Hello Linux[jkim@localhost chap16]$
```

gcc 옵션

-E	전처리를 실행하고 컴파일을 중단하게 한다.
-c	소스 파일을 컴파일만 하고, 링크를 수행하지 않고 오브젝트 파일을 생성한다.
-o	바이너리 형식의 출력 파일 이름을 지정하는데, 지정하지 않으면 a.out라는 기본 이름이 적용된다.
-I	헤더 파일을 검색하는 디렉토리 목록을 추가한다.
-L	라이브러리 파일을 검색하는 디렉토리 목록을 추가한다.
-l	라이브러리 파일을 컴파일시 링크한다.
-g	바이너리 파일에 표준 디버깅 정보를 포함시킨다.
-ggdb	바이너리 파일에 GNU 디버거인 gdb만이 이해할 수 있는 많은 디버깅 정보를 포함시킨다.
-O	컴파일 코드를 최적화시킨다.
-ON	최적화 N 단계를 지정한다.
-DFOO=RAR	명령라인에서 BAR의 값을 가지는 FOO라는 선행 처리기 매크로를 정의한다.
-static	정적 라이브러리에 링크한다.
-ansi	표준과 충돌하는 GNU 확장안을 취소하며 ANSI/ISO C 표준을 지원한다. 이 옵션은 ANSI 호환 코드를 보장하지 않는다.
-traditional	과거 스타일의 함수 정의 형식과 같이 전통적인 K&R(Kernighan and Ritchie) C 언어 형식을 지원한다.
-MM	make 호환의 의존성 목록을 출력한다.
-V	컴파일의 각 단계에서 사용되는 명령을 보여준다.

-o 옵션

```
-o 옵션
기능
        생성되는 출력 파일 이름을 지정한다.
기본형
        gcc -o 출력파일이름 소스파일이름
```

```
[jkim@localhost chap16]$ gcc -o file file.c↵
[jkim@localhost chap16]$
```

```
[jkim@localhost chap16]$ gcc file.c -o file↵
[jkim@localhost chap16]$
```

```
[jkim@localhost chap16]$ ./file↵
Hello Linux[jkim@localhost chap16]$
```

-E 옵션

```
-E 옵션
기능
        전처리까지만 실행하고 결과를 화면에 출력한다.
기본형
        gcc -E 소스파일이름
```

```
[jkim@localhost chap16]$ gcc -E file.c↵
# 1 "file.c"
# 1 "/usr/include/stdio.h" 1 3
# 27 "/usr/include/stdio.h" 3
# 1 "/usr/include/features.h" 1 3
# 283 "/usr/include/features.h" 3
    ⋮
extern void funlockfile (FILE *__stream) ;

[jkim@localhost chap16]$
```

```
[jkim@localhost chap16]$ gcc -E file.c -o file.i↵
[jkim@localhost chap16]$
```

-c 옵션

```
-c 옵션
기능
        전처리, 컴파일, 어셈블까지 실행하여 오브젝트 파일을 생성한다.
기본형
        gcc -c 소스파일이름
```

```
[jkim@localhost chap16]$ gcc -c file.c↵
[jkim@localhost chap16]$ ls↵
file.c  file.o
[jkim@localhost chap16]$
```

```
[jkim@localhost chap16]$ gcc file.o↵
[jkim@localhost chap16]$ ls↵
a.out*  file.c  file.o
[jkim@localhost chap16]$
```

-c 옵션

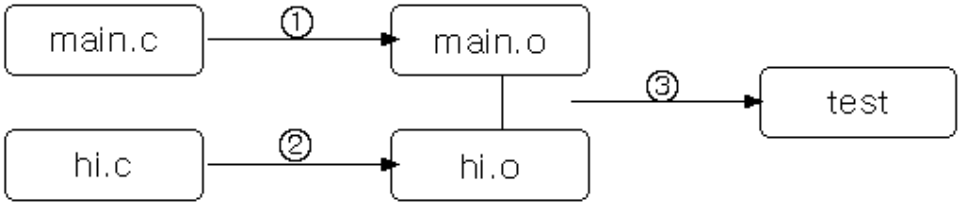
분리 컴파일 ([프로그램 16-2] [프로그램 16-3])

```
[jkim@localhost chap16]$ gcc main.c hi.c -o test↵
[jkim@localhost chap16]$
```

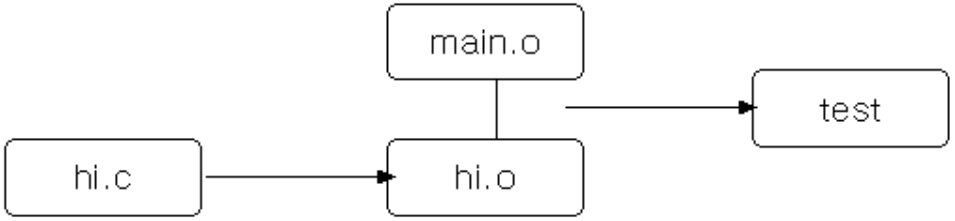
```
[jkim@localhost chap16]$ gcc main.c -o test↵
/tmp/ccFpZHdU.o: In function `main':
/tmp/ccFpZHdU.o(.text+0x7): undefined reference to `hi'
collect2: ld returned 1 exit status
[jkim@localhost chap16]$ gcc hi.c -o test↵
/usr/lib/gcc-lib/i386-redhat-linux/2.96/../../../crt1.o: In function `_start':
/usr/lib/gcc-lib/i386-redhat-linux/2.96/../../../crt1.o(.text+0x18): undefined reference to `main'
collect2: ld returned 1 exit status
[jkim@localhost chap16]$
```

-c 옵션

```
[jkim@localhost chap16]$ gcc -c main.c↵    … ①
[jkim@localhost chap16]$ gcc -c hi.c↵    … ②
[jkim@localhost chap16]$ gcc main.o hi.o -o test↵    … ③
[jkim@localhost chap16]$
```



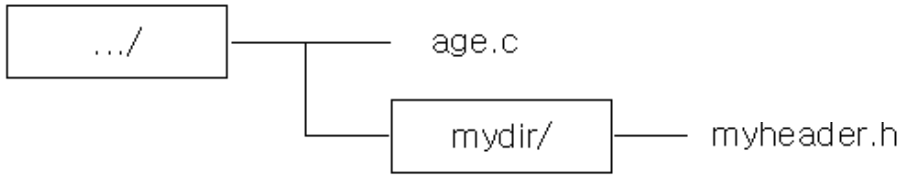

```
$ gcc -c hi.c↵
$ gcc main.o hi.o -o test↵
$
```



-I 옵션

```
-I 옵션
기능
        표준 디렉토리가 아닌 위치에 있는 헤더 파일의 디렉토리를 지정한다.
기본형
        gcc 소스파일이름 -I디렉토리이름
```

■ [프로그램 16-4], [프로그램 16-5]

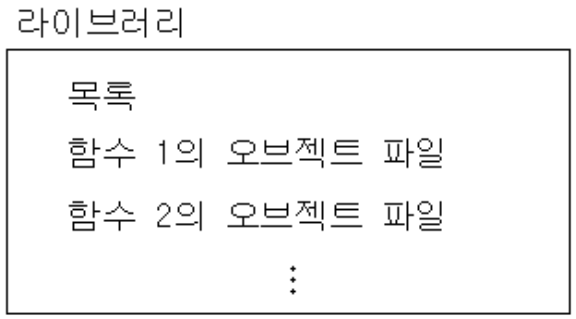



```
[jkim@localhost chap16]$ gcc age.c↵
age.c:2:22: myheader.h: No such file or directory
[jkim@localhost chap16]$
```

```
[jkim@localhost chap16]$ gcc age.c -Imydir↵
[jkim@localhost chap16]$
```

라이브러리란?

- 자주 사용되는 유용한 함수에 대한 오브젝트 파일을 모아 둔 것으로 함수 목록도 포함




```
[jkim@localhost chap16]$ cd /usr/lib↵
[jkim@localhost lib]$ ls↵
        ⋮
libc.a          libm.a
        ⋮
[jkim@localhost lib]$
```

라이브러리 생성하기

- [프로그램 16-6], [프로그램 16-7]

```
[jkim@localhost mylib]$ gcc -c plus.c minus.c↵
[jkim@localhost mylib]$
```

라이브러리 생성

```
[jkim@localhost mylib]$ ar r libmy.a plus.o minus.o↵
[jkim@localhost mylib]$ ls↵
libmy.a  minus.c  minus.o  plus.c  plus.o
[jkim@localhost mylib]$
```

목록 추가

```
[jkim@localhost mylib]$ ar s libmy.a↵
[jkim@localhost mylib]$
```

-l 옵션

```
-l 옵션
기능
        표준 라이브러리가 아닌 라이브러리를 지정한다.
기본형
        gcc 소스파일이름 -l라이브러리이름
```

■ [프로그램 16-8]

```
[jkim@localhost chap16]$ gcc 16_8.c -lm↵
[jkim@localhost chap16]$
```

■ [프로그램 16-9]

```
[jkim@localhost chap16]$ gcc 16_9.c -lmy↵
/usr/bin/ld: cannot find -lmy
collect2: ld returned 1 exit status
[jkim@localhost chap16]$
```

-L 옵션

```
-L 옵션
기능
        사용할 라이브러리의 위치를 지정한다.
기본형
        gcc 소스파일이름 -L라이브러리위치
```

```
[jkim@localhost chap16]$ gcc 16_9.c -lmy -Lmylib↵
[jkim@localhost chap16]$
```

디버깅 관련 옵션

-g, -ggdb 옵션
기능
　　디버깅 정보를 삽입한다.
기본형
　　gcc -g 소스파일이름
　　gcc -ggdb 소스파일이름

■ -g 옵션의 단계

-g1	역추적 스택 덤프 생성에 필요한 정보를 포함하지만 지역변수, 문장 번호를 위한 디버깅 정보는 삽입하지 않는다.
-g2	확장기호 테이블, 문장 번호, 지역과 외부변수에 대한 디버깅 정보를 삽입한다.
-g3	-g2 옵션의 디버깅 정보와 모든 매크로 정의를 삽입한다.

최적화 옵션

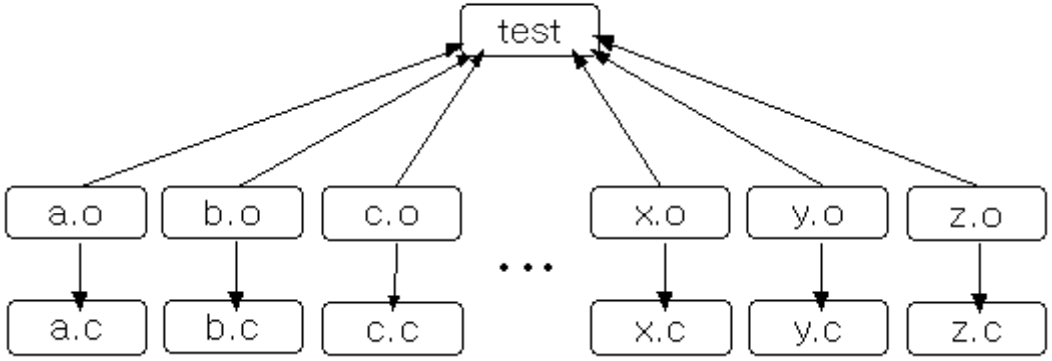
-O 옵션 기능 코드를 최적화시킨다. 기본형 gcc -O 소스파일이름

■ -O 옵션의 단계

-O1	-O 옵션과 같은 단계의 옵션으로 최소한으로 스레드 분기 동작 횟수를 줄이고, 호출된 각 함수 반환 시 스택에 인수를 모아 두었다 동시에 꺼내게 해 준다.
-O2	-O1 단계의 최적화와 함께 프로세서가 다른 지시어의 결과를 기다리거나 캐시, 주 메모리에서 데이터를 기다리는 동안 컴파일러가 지시어를 실행하도록 한다. 컴파일 시간이 더 오래 걸리지만, 수정된 코드는 더 최적화되어 실행이 빨라진다.
-O3	-O2 단계의 모든 최적화와 루프 해체, 그 밖의 프로세서 전용 특징을 포함한다.

make란?

- 다수의 소스 파일로 구성된 큰 프로그램 중 어떤 파일이 변경되어 재 컴파일이 필요한지 판단해 프로그램 재구성 작업을 효율적으로 수행하는 역할을 한다.



make 파일

- 어떤 일을 해야 하는지 make에 알려주어야 하는 정보를 담고 있는 파일
- 형태

```
대상(target): 대상에 의존되는 파일1 [파일2 ...]
    명령(command)
```

대상

■ make가 궁극적으로 생성하는 것으로, 일반적으로 오브젝트 파일이나 실행 파일이 된다.

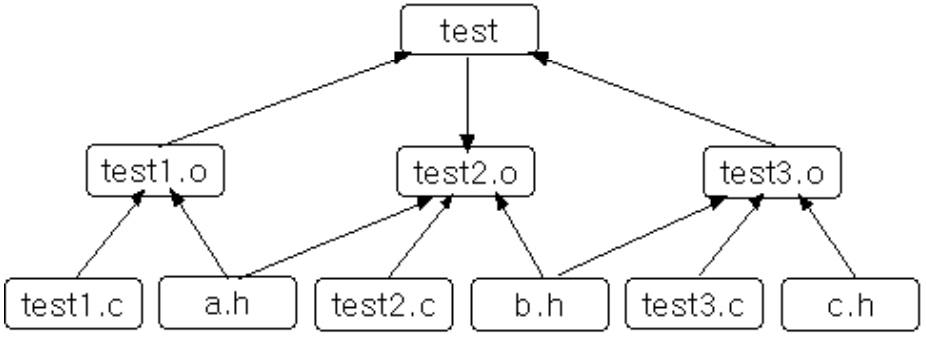
```
test: test.c
	gcc test.c -o test
```

```
clean:
	rm test.o
```

의존성

■ 대상과 대상을 생성하는 데 필요한 소스 파일의 관계

```
test: test1.o test2.o test3.o    … ①
test1.o: test1.c a.h    … ②
test2.o: test2.c a.h b.h    … ③
test3.o: test3.c b.h c.h    … ④
```



명령

- 반드시 탭 문자로 시작

```
test: test1.o test2.o test3.o
[  탭  ]gcc -o test test1.o test2.o test3.o
```

make 예제

■ [프로그램 17-1, 2, 3]

[makefile]

```
01 test: test1.o test2.o test3.o
02       gcc -o test test1.o test2.o test3.o
03 test1.o: test1.c a.h
04       gcc -c test1.c
05 test2.o: test2.c a.h b.h
06       gcc -c test2.c
07 test3.o: test3.c b.h c.h
08       gcc -c test3.c
```

```
[jkim@localhost chap17]$ make↵
gcc -c test1.c
gcc -c test2.c
gcc -c test3.c
gcc -o test test1.o test2.o test3.o
[jkim@localhost chap17]$
```

make 예제

```
[jkim@localhost chap17]$ make test1.o↵
gcc -c test1.c
[jkim@localhost chap17]$ make test2.o↵
gcc -c test2.c
[jkim@localhost chap17]$ make test3.o↵
gcc -c test3.c
[jkim@localhost chap17]$ make test↵
gcc -o test test1.o test2.o  test3.o
[jkim@localhost chap17]$
```

```
[jkim@localhost chap17]$ make↵
make: `test' is up to date.
[jkim@localhost chap17]$
```

```
[jkim@localhost chap17]$ touch b.h↵
[jkim@localhost chap17]$ make↵
gcc -c test2.c
gcc -c test3.c
gcc -o test test1.o test2.o  test3.o
[jkim@localhost chap17]$
```

가짜 대상

[makefile]

```
01 test: test1.o test2.o test3.o
02         gcc -o test test1.o test2.o test3.o
03 test1.o: test1.c a.h
04           gcc -c test1.c
05 test2.o: test2.c a.h b.h
06         gcc -c test2.c
07 test3.o: test3.c b.h c.h
08         gcc -c test3.c
09 clean:
10     rm test1.o test2.o test3.o
```

```
[jkim@localhost chap17]$ make clean↵
rm test1.o test2.o test3.o
[jkim@localhost chap17]$
```

주석

[makefile]

```
01 # make 파일 주석문 사용 예제
02 # 실행 파일 생성
03 test: test1.o test2.o test3.o # test 파일 의존성 목록
04       gcc -o test test1.o test2.o test3.o
05 # 오브젝트 파일 생성
06 test1.o: test1.c a.h
07       gcc -c test1.c
08 test2.o: test2.c a.h b.h
09       gcc -c test2.c
10 test3.o: test3.c b.h c.h
11       gcc -c test3.c
12 # 가짜 대상
13 clean:
14       rm test1.o test2.o test3.o
```

```
test1.o: test1.c a.h
     gcc -c test1.c  # 이곳에는 주석문을 쓰지 않는 것이 좋다.
```

자동 의존 관계 생성

```
01 test: test1.o test2.o test3.o
02       gcc -o test test1.o test2.o test3.o
03 test1.o: test1.c a.h
04       gcc -c test1.c
05 test2.o: test2.c a.h b.h
06       gcc -c test2.c
07 test3.o: test3.c b.h c.h
08       gcc -c test3.c
09 clean:
10       rm test1.o test2.o test3.o
11 # 추가 부분
12 dep:
13       gccmakedep test1.c test2.c test3.c
```

```
[jkim@localhost chap17]$ make dep↵
gccmakedep test1.c test2.c test3.c
[jkim@localhost chap17]$
```

긴 명령어 쓰기

```
test: test1.o test2.o test3.o test4.o test5.o test6.o test7.o
      test8.o test9.o    … 이 부분을 인지하지 못한다.
      gcc -o test test1.o test2.o test3.o test4.o test5.o test6.o
                test7.o test8.o test9.o    … 이 부분을 인지하지 못한다.
```

```
test: test1.o test2.o test3.o test4.o test5.o test6.o test7.o₩
      test8.o test9.o
      gcc -o test test1.o test2.o test3.o test4.o test5.o test6.o₩
                test7.o test8.o test9.o
```

여러 개의 셸 명령 사용

```
test1.o: test1.c a.h
	cd testdir
	gcc -c test1.c
	mv *.o /home/root
```

```
test1.o: test1.c a.h
	cd testdir ; gcc -c test1.c ; mv *.o /home/root
```

```
test1.o: test1.c a.h
	cd testdir ;₩
	gcc -c test1.c ;₩
	mv *.o /home/root
```

명령 실패 무시

```
01 test: test1.o test2.o test3.o
02    gcc -o test test1.o test2.o test3.o
03 test1.o: test1.c a.h
04    gcc -c test1.c
05 test2.o: test2.c a.h b.h
06    gcc -c test2.c
07 test3.o: test3.c b.h c.h
08    gcc -c test3.c ;₩
09    cp t3.o test4.o
10 clean:
11    rm test1.o test2.o test3.o
```

매크로 정의

```
M_NAME = value
```

```
$(M_NAME)
```

```
01 OBJF = test1.o test2.o test3.o
02 test: $(OBJF)
03    gcc -o test $(OBJF)
04 test1.o: test1.c a.h
05    gcc -c test1.c
06 test2.o: test2.c a.h b.h
07    gcc -c test2.c
08 test3.o: test3.c b.h c.h
09    gcc -c test3.c
10 clean:
11    rm $(OBJF)
```

내부 매크로

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

```
01 OBJF = test1.o test2.o test3.o
02 test: $(OBJF)
03    gcc -o $@ $^
04 test1.o: test1.c a.h
05    gcc -c $<
06 test2.o: test2.c a.h b.h
07    gcc -c $*.c
08 test3.o: test3.c b.h c.h
09    gcc -c $*.c
10 clean:
11    rm $(OBJF)
```

매크로 수정

```
OB = test1.o test2.o
OBJF = $(OB) test3.o
```

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

```
[jkim@localhost chap17]$ make↵
makefile:3: *** Recursive variable `OBJF' references itself (eventually).  Stop.
[jkim@localhost chap17]$
```

```
OBJF = test1.o test2.o
OBJF := $(OBJF) test3.o
```

매크로 치환

```
$(M_NAME:old=new)
```

```
OBJF = test1.o test2.o test3.o
SRCS = $(OBJF:.o=.c)
```

암시적 규칙

■ make 내에 미리 정의된 규칙을 이용해 make 파일을 단순화시키는 규칙

```
01 OBJF = test1.o test2.o test3.o
02 test: $(OBJF)
03    gcc -o $@ $(OBJF)
04 clean:
05    rm $(OBJF)
```

```
[jkim@localhost chap17]$ make↵
cc     -c -o test1.o test1.c
cc     -c -o test2.o test2.c
cc     -c -o test3.o test3.c
gcc -o test test1.o test2.o test3.o
[jkim@localhost chap17]$
```

```
[jkim@localhost chap17]$ make -p↵
      ⋮
[jkim@localhost chap17]$
```

접미사 규칙

```
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
```

```
.c.o:    … ①
	gcc -c $< $(CFLAGS)    … ②
```

```
01 OBJF = test1.o test2.o test3.o
02 test: $(OBJF)
03    gcc -o $@ $(OBJF)
04 .c.o:
05    gcc -c $(CFLAGS) $<
06 clean:
07    rm $(OBJF)
```

```
[jkim@localhost chap17]$ make CFLAGS="-g"↵
    ⋮
[jkim@localhost chap17]$
```

패턴 규칙

```
01 OBJF = test1_d.o test2_d.o test3_d.o
02 test: $(OBJF)
03    gcc -o $@ $(OBJF)
04 %_d.o: %.c
05    gcc -c -g $< -o $@
06 clean:
07    rm $(OBJF)
```

```
[jkim@localhost chap17]$ make↵
gcc -c -g test1.c -o test1_d.o
gcc -c -g test2.c -o test2_d.o
gcc -c -g test3.c -o test3_d.o
gcc -o test test1_d.o test2_d.o test3_d.o
[jkim@localhost chap17]$
```

make 옵션

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

-f 옵션

```
-f 옵션
기능
        표준 파일 외의 파일을 실행한다.
기본형
        make -f 파일이름
```

```
[jkim@localhost chap17]$ make -f m_file↵
```

-n, -W 옵션

```
-n 옵션
기능
        실행하지는 않으면서 실행해야 할 명령을 출력한다.
기본형
        make -n
```

```
-W 옵션
기능
        파일이 변경된 것처럼 동작한다.
기본형
        make -W 파일이름
```

```
[jkim@localhost chap17]$ make↵
gcc -c test1.c
gcc -c test2.c
gcc -c test3.c
gcc -o test test1.o test2.o test3.o
[jkim@localhost chap17]$ make -n -Wtest1.c↵
gcc -c test1.c
gcc -o test test1.o test2.o test3.o
[jkim@localhost chap17]$
```

-s 옵션

```
-s 옵션
기능
        실행하는 명령을 출력하지 않고 실행한다.
기본형
        make -s
```

```
[jkim@localhost chap17]$ make -s↵
[jkim@localhost chap17]$
```

-r 옵션

```
-r 옵션
기능
        모든 내장 규칙을 사용할 수 없게 한다.
기본형
        make -r
```

```
01 OBJF = test1.o test2.o test3.o
02 test: $(OBJF)
03    gcc -o $@ $(OBJF)
04 clean:
05    rm test $(OBJF)
```

```
[jkim@localhost chap17]$ make -r↵
make: *** No rule to make target `test1.o', needed by `test'.  Stop.
[jkim@localhost chap17]$
```

-d 옵션

```
-d 옵션
기능
        디버깅 정보를 출력한다.
기본형
        make -d
```

```
[jkim@localhost chap17]$ make -d↵

    ⋮
Got a SIGCHLD; 1 unreaped children.
Reaping winning child 0x08071238 PID 10549
Removing child 0x08071238 PID 10549 from chain.
Successfully remade target file `test'.
[jkim@localhost chap17]$
```

-k 옵션

```
-k 옵션
기능
        명령에 실패해도 계속 동작한다.
기본형
        make -k
```

```
01 OBJF = test1.o test2.o test3.o
02 test : $(OBJF)
03    gcc -o $@ $^
04 test1.o : test1.c a.h
05    gcc -c $<
06 test2.o : test2.c a.h b.h
07    gcc -c $<
08 test3.o : test3.c b.h c.h
09    cp t3.c t1.c ;₩
10    gcc -c $<
11 clean:
12    rm $(OBJF) test
```

```
[jkim@localhost chap17]$ make -k↵
    ⋮
cp: cannot stat `t3.c': No such file or directory
gcc -o test test1.o test2.o test3.o
[jkim@localhost chap17]$
```

gdb 실행하고 종료하기

■ [프로그램 18-1]

```
[jkim@localhost chap18]$ gcc -g debug1.c -o debug1↵
[jkim@localhost chap18]$
```

```
$ gdb 실행파일이름 [코어덤프파일이름]↵
```

```
[jkim@localhost chap18]$ gdb debug1↵
GNU gdb Red Hat Linux 7.x (5.0rh-15hl) (MI_OUT)
    ⋮
This GDB was configured as "i386-redhat-linux"...
(gdb) _
```

```
(gdb) quit↵
[jkim@localhost chap18]$
```

디렉토리 지정하기

```
[jkim@localhost chap18]$ gdb -d source debug1↵
[jkim@localhost chap18]$
```

도움말 보기

```
(gdb) n⇆
next            nexti           ni              nosharedlibrary
(gdb) _
```

```
(gdb) help↵
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
    ⋮
Command name abbreviations are allowed if unambiguous.
(gdb) _
```

```
(gdb) help list↵
List specified function or line.
With no argument, lists ten more lines after or around previous listing.
    ⋮
With two args if one is empty it stands for ten lines away from the other arg.
(gdb) _
```

소스 파일 내용 출력하기

```
(gdb) list↵
1       #include <stdio.h>
2       int sum(int i);
3
4       main()
5       {
6           printf("%d₩n", sum(10));
7       }
8
9       int sum(int i)
10      {
(gdb) _
```

```
(gdb) ↵
11          return i+sum(i-1);
12      }
(gdb) _
```

소스 파일 내용 출력하기

```
(gdb) list 4, 7↵
4       main()
5       {
6           printf("%d₩n", sum(10));
7       }
(gdb) _
```

```
(gdb) list 함수이름↵
```

```
(gdb) list↵
1       debug1.c: No such file or directory.
        in debug1.c
(gdb) _
```

프로그램 실행시키기

```
(gdb) run↵
Starting program: /home/jkim/book/linux/chap18/debug1

Program received signal SIGSEGV, Segmentation fault.
0x08048492 in sum (i=-349476) at debug1.c:11
11          return i+sum(i-1);
(gdb) _
```

■ [프로그램 18-2]

```
[jkim@localhost chap18]$ gcc -g debug2.c -o debug2↵
[jkim@localhost chap18]$ gdb debug2↵
    ⋮
(gdb) run hello linux programming↵
Starting program: /home/jkim/book/linux/chap18/debug2 hello linux programming
    ⋮
Program exited with code 04.
(gdb) _
```

변수 다루기

```
(gdb) whatis i↵
type = int
(gdb) print i↵
$1 = -349476
(gdb) _
```

```
(gdb) set variable i=5↵
(gdb) print i↵
$2 = 5
(gdb) _
```

정지점 설정하기

```
break 문장번호
break 함수이름
break 문장번호 또는 함수이름 if 조건
```

```
(gdb) break 11 if i==0
```

```
[jkim@localhost chap18]$ gcc -g debug3.c -o debug3↵
[jkim@localhost chap18]$ gdb debug3↵
    ⋮
(gdb) _
```

```
(gdb) break sum_all↵
Breakpoint 1 at 0x804850e: file debug3.c, line 22.
(gdb) _
```

정지점 설정하기

```
(gdb) run↵
Starting program: /home/jkim/book/linux/chap18/debug3
Input integer ==> 9
Input integer ==> 7
Input integer ==> 5
Input integer ==> 3
Input integer ==> 1

Breakpoint 1, sum_all (c=0) at debug3.c:22
22          if (c>=5)
(gdb) _
```

```
(gdb) continue↵
Continuing.

Breakpoint 1, sum_all (c=0) at debug3.c:22
22          if (c>=5)
(gdb) _
```

```
(gdb) break 10↵
Breakpoint 2 at 0x8048496: file debug3.c, line 10.
(gdb) break 24↵
Breakpoint 3 at 0x804851c: file debug3.c, line 24.
(gdb) _
```

정지점 확인과 제거하기

```
(gdb) info break↵
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x0804850e in sum_all at debug3.c:22
2   breakpoint     keep y   0x08048496 in main at debug3.c:10
3   breakpoint     keep y   0x0804851c in sum_all at debug3.c:24
(gdb) _
```

```
(gdb) delete 2↵
(gdb) info break↵
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x0804850e in sum_all at debug3.c:22
3   breakpoint     keep y   0x0804851c in sum_all at debug3.c:24
(gdb) _
```

```
(gdb) delete↵
Delete all breakpoints? (y or n) y↵
(gdb) info break↵
No breakpoints or watchpoints.
(gdb) _
```

단계별로 실행하기

- [프로그램 18-4]
- step

```
(gdb) break 9↵
Breakpoint 1 at 0x8048466: file debug4.c, line 9.
(gdb) run↵
Starting program: /home/jkim/book/linux/chap18/debug4

Breakpoint 1, main () at debug4.c:9
9               s = sum(10);
(gdb) step↵
sum (i=10) at debug4.c:15
15              if (i <=0)
(gdb) step↵
18                  return i+sum(i-1);
(gdb)
```

단계별로 실행하기

- next

```
(gdb) break 9
Breakpoint 1 at 0x8048466: file debug4.c, line 9.
(gdb) run
Starting program: /home/jkim/book/linux/chap18/debug4

Breakpoint 1, main () at debug4.c:9
9           s = sum(10);
(gdb) next
10          printf("%d₩n", s);
(gdb) next
55
11      }
(gdb)
```

- 여러 단계 실행

```
(gdb) step 5
(gdb) next 7
```