



# 리눅스 프로그래밍

---

MCALAB  
작성자 : 양평우  
정용희

# 프로그래밍 순서와 Tool

1. 계획
  - 어떤 방법으로 프로그램이 동작할 지 구상
2. 코딩
  - 컴퓨터에 코드를 작성하는 과정 (*vi*, *vim*, *emacs* 등을 사용)
  - 편하게 코딩하기 위해 *Nano*(우분투) 에디터를 사용
3. Compile
  - 코드를 실행 가능한 파일로 만든다. (*gcc*를 사용)
  - 많은 코드를 컴파일 하거나, *dependancy*가 있는 프로그램일 경우에는 *make*를 이용해 편하게 할 수 있다.
4. 컴파일 에러를 잡는다.
  - Compile시에 나는 에러메시지를 보고 없애 줍시다.
  - 다시 3으로 가서 컴파일
5. 실행
  - 작성한 프로그램이 잘 돌아가는지 테스트
  - 잘 돌아가면 끝!
  - 원하던 대로 돌아가지 않는다면 디버깅한다.
6. 디버깅
  - 잘 돌아가지 아니하면 버그를 찾아야 한다. (*gdb*, *ddd* 등 사용)

# Nano 에디터

## □ 사용법

### ■ # nano \*(파일이름).c

```
root@yongza-desktop:/home/yongza/study/code# nano ex1.c
```

```
GNU nano 2.2.2          파일 : ex1.c

#include <stdio.h>

void main(void)
{
    printf("Hello World\n");
}
```

# gcc

- ❑ C file → executable file
- ❑ Compile & Link
  - Compile (in C)
    - ❑ C program을 machine code로 번역
    - ❑ Object code를 생성
  - Link
    - ❑ 여러 object code들을 연결!
- ❑ *gcc*
  - GNU C Compiler

# gcc

## □ 사용법

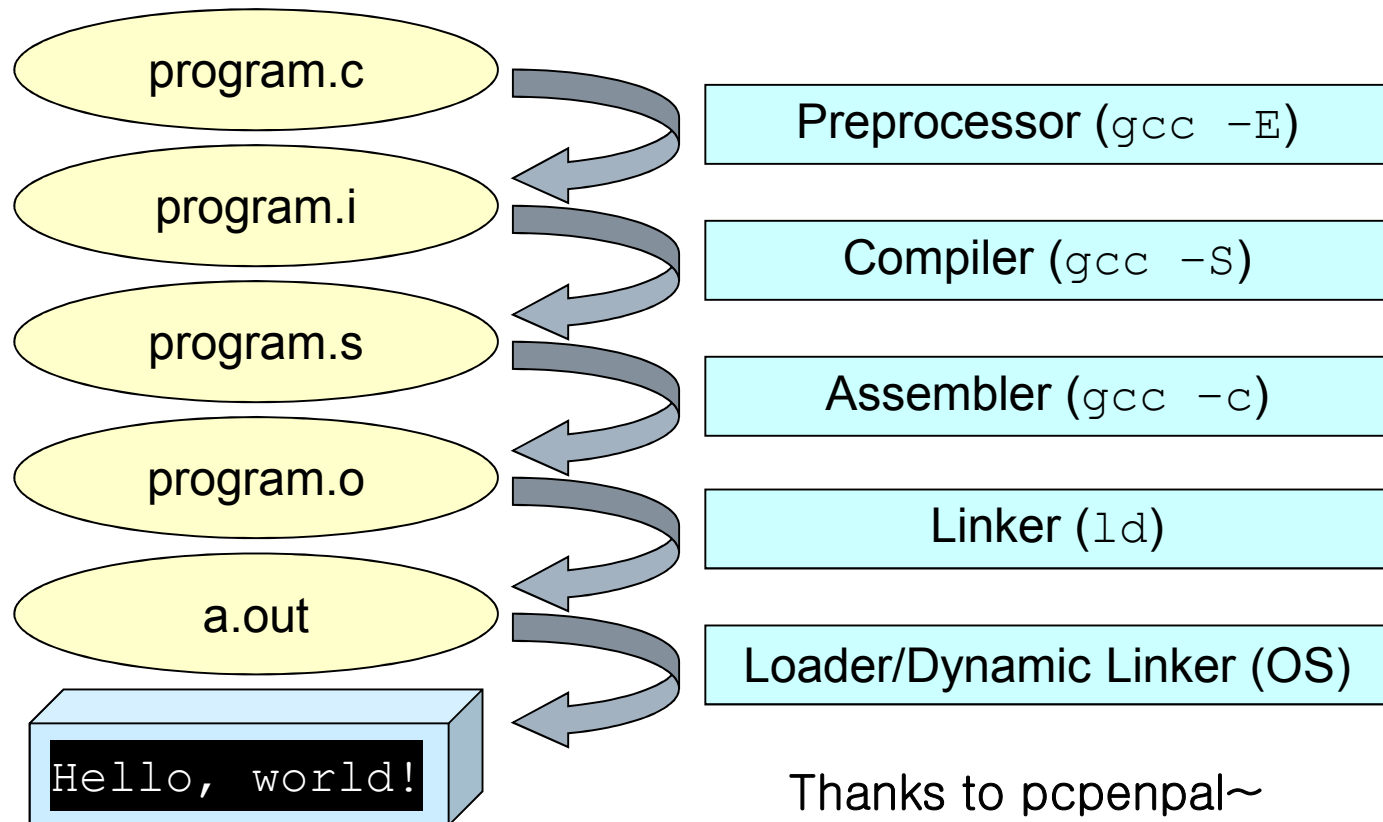
- `gcc [option] <source files>`

## □ 옵션

- `-Wall`: 하찮은 warning들 까지 모두 출력
- `-ansi`: ANSI C 표준에 의거해서 컴파일
- `-o <filename>`: 컴파일 후 생성될 파일 이름 지정
  - 생략 시 `a.out`으로 생성
- `-O<n>`:  $0 < n < 7$  에 해당하는  $n$  의 단계로 최적화
  - Ex) `gcc -O2 temp.c`
  - 높을 수록 많이 최적화
  - 최적화단계를 높일수록 컴파일 시간은 늦어짐
- `-v`: 컴파일 과정을 보여줌

# gcc

- 컴파일에서 실행까지의 세부 단계



# Gcc : Case by Case

- 기본
  - `$gcc test1.c -> "a.out"` 실행파일 생성
- 실행파일 이름을 지정하고 싶을 때(-o)
  - `$gcc -o test test1.c`
- 오브젝트 파일로만 컴파일하고 링크를 생략할 때(-c)
  - `$gcc -c test1.c`
- 여러 개의 소스파일을 컴파일하여 하나의 실행파일 만들때
  - `$gcc -o test test1.c test2.c`
  - 이것은 다음과 동일
    - `$gcc -c test1.c test2.c`
    - `$gcc -o test test1.o test2.o`
- 코드를 최적화하고 싶을 때
  - `$gcc -O2 -o test test1.c`

# Gcc : Case by Case

- ❑ 디버깅 기능을 사용하고 싶을 때(-g)
  - `$gcc -g -o test test1.c`
- ❑ 수학 라이브러리(math.h)를 사용하고 싶을 때(-l)
  - `$gcc -o test test1.c -lm`
    - ❑ /usr/lib 디렉토리에 있는 기본 라이브러리를 사용한다
    - ❑ libxxx.a 의 형태로 사용하며, 라이브러리 링크시 -lxxx
    - ❑ 예) /usr/lib/libgdbm.a 를 사용한다면, -lgdbm
    - ❑ 여러 개 사용시 -lxxx1 -lxxx2 형태로 사용
- ❑ Header 경로에 특정 디렉토리를 추가할때(-I)
  - `$gcc -I ../include -o test test1.c`
- ❑ 라이브러리 경로에 특정 디렉토리를 추가할때(-L)
  - `$gcc -I ../include -L ../lib -o test test1.c`

# gcc : 다중 파일 c 프로그래밍의 예

- Function들로만 구성된 파일
  - 헤더파일 만들기(myfunc.h)

```
extern void sayHello(void);
```

- Function 파일(myfunc.c)

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

void sayHello(void)
{
    printf(" hello! I call this func in other file \n");
}
```

- Main 파일(myhello.c)

```
#include <stdio.h>
void main()
{
    sayHello();
}
```

# gcc : 다중 파일 c 프로그래밍의 예

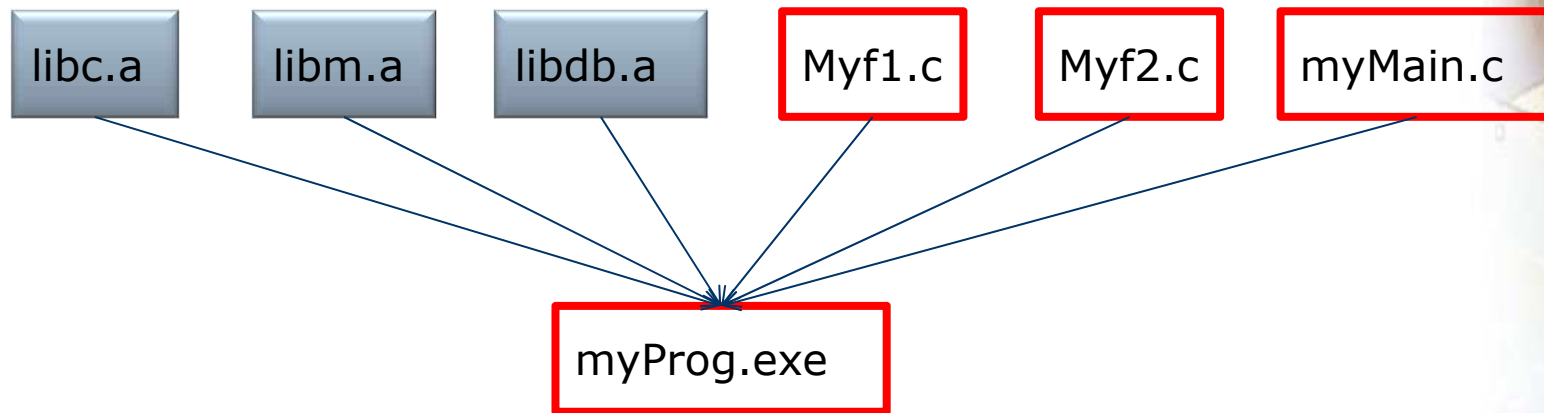
## □ 프로그램의 컴파일

■ `$gcc -o myhello myhello.c myfunc.c`

```
root@ubuntu:/home/yongza/study# gcc -o myhello.exe myhello.c myfunc.c
root@ubuntu:/home/yongza/study# ./myhello.exe
Hello! i can this funs func in other file
```

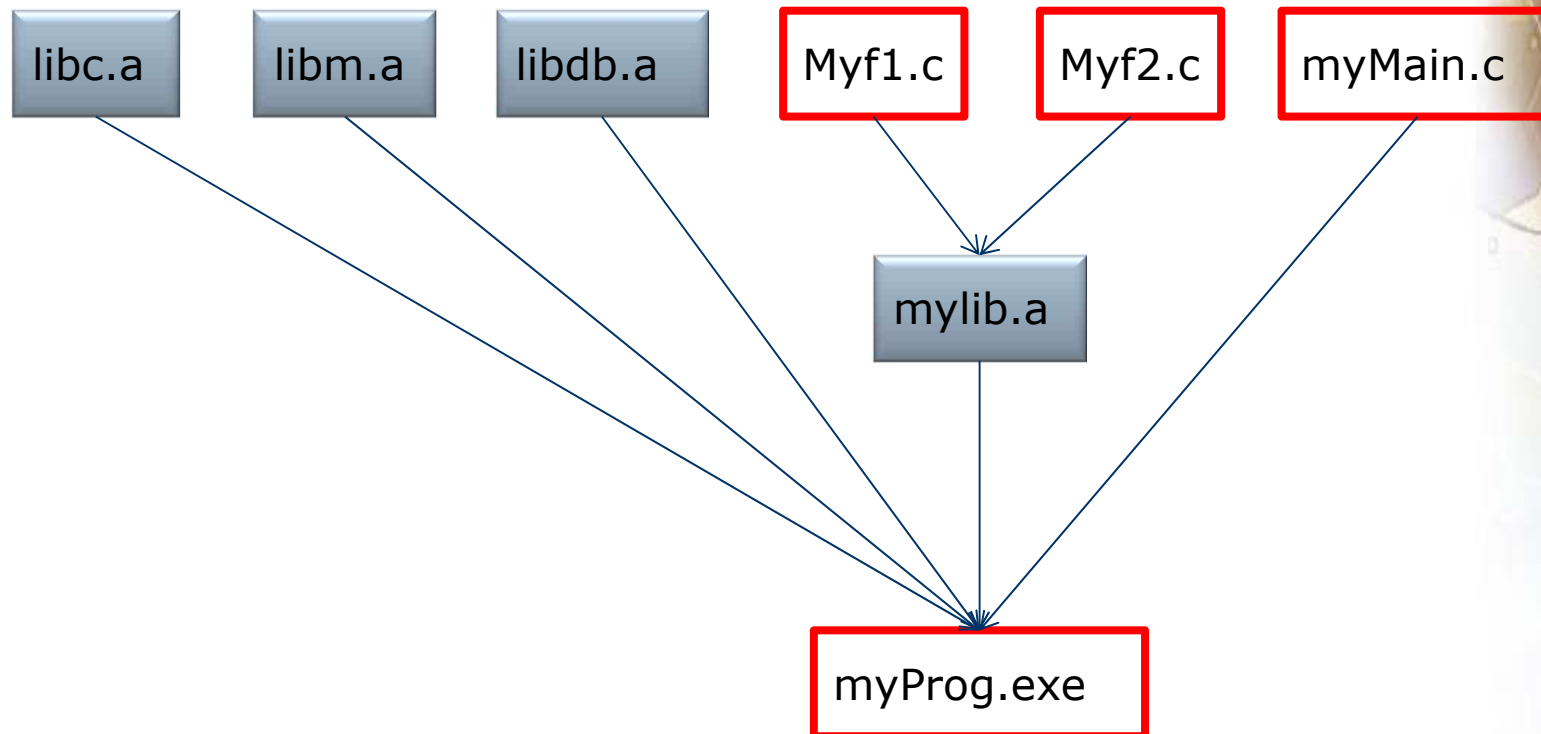
# gcc : 라이브러리 만들어 재 사용하기

## □ 실행파일 컴파일(일반)



# gcc : 라이브러리 만들어 재 사용하기

- 자신만의 라이브러리 만들어 컴파일하기



# gcc : 라이브러리 만들고 사용하기

- ❑ 라이브러리에 포함될 파일을 object파일로 컴파일
  - `$gcc -c myfunc.c`
    - ❑ `myfunc.o` 가 생성됨
- ❑ 라이브러리 파일을 생성하기(ar을 사용)
  - `$ar r libmylib.a myfunc.o`
    - ❑ `r` : 새로운 라이브러리 파일을 생성하거나 대체
  - `$ar s libmylib.a`
    - ❑ `s` : 라이브러리안에 인덱스 파일을 생성함
- ❑ 라이브러리 안의 오브젝트들 보기
  - `$ar t libmylib.a`
    - ❑ `$ar t /usr/lib/libm.a`

# gcc : 라이브러리 만들고 사용하기

- 라이브러리를 이용한 프로그램 만들기
  - `$gcc -o myhello2.exe myhello.c -lmylib -L .`

```
root@ubuntu:/home/yongza/study# gcc -c myfunc.c
root@ubuntu:/home/yongza/study# ls
make1 myfunc.c myfunc.h myfunc.o myhello.c myhello.exe
root@ubuntu:/home/yongza/study# ar r libmylib.a myfunc.o
ar: creating libmylib.a
root@ubuntu:/home/yongza/study# ar s libmylib.a
root@ubuntu:/home/yongza/study# ar t libmylib.a myfunc.o
myfunc.o
root@ubuntu:/home/yongza/study# gcc -o myhello2.exe myhello.c -lmylib
-L .
root@ubuntu:/home/yongza/study# ls
libmylib.a myfunc.c myfunc.o myhello.exe myhello2.exe
make1      myfunc.h myhello.c
root@ubuntu:/home/yongza/study# ./myhello2.exe
Hello! i can this func in other file
```

# make

- ❑ program group을 'make'하는 도구
- ❑ 장점
  - 필요한 부분만 recompile을 자동으로 처리
  - 프로그램들 간의 상관 및 의존 관계를 자동으로 처리
  - clean-up 및 build-all 과정을 단순화
- ❑ 사용법
  - `$make`
    - ❑ 현재 디렉토리에 파일이름이 Makefile 인 파일을 이용한 make
  - `$make -f MyMakefile`
    - ❑ 특정 Makefile을 지정한 make

# make

## □ Makefile

- Makefile은 다음과 같은 형태로 구성됨
  - 설정
  - 목표 (target) : 의존 관계 (dependancy)
    - ◆ 명령 (rule)
- 예) myhello 프로그램의 컴파일
  - Makefile의 내용을 다음과 같이 작성

주의: 명령어 앞에는  
항상 **tab**이 있어야함!

```
CC = gcc
myhello3.out: myhello.c myfunc.c
>      $(CC) -o myhello3.out myhello.c myfunc.c
```

## ■ 실행

- \$make
- \$make myhello3.out



```
developer@sandbox: ~/prog
파일(F) 편집(E) 보기(V) 터미널(T) 도움말(H)

developer@sandbox:~/prog$ make
gcc -o myhello3.out myhello.c myfunc.c
developer@sandbox:~/prog$ ls
Makefile  myfunc.c  myfunc.o  myhello.c  myhello3.exe
libmylib.a myfunc.h  myhello   myhello2.exe myhello3.out
developer@sandbox:~/prog$
```

# make

## ❑ Makefile에서 Target의 사용

```
hello: myhello.o myfunc.o
    gcc -o say_linux myhello.o myfunc.o
myhello.o: myhello.c
    gcc -c myhello.c
myfunc.o: myfunc.c
    gcc -c myfunc.c
```

## ■ Make

- ❑ \$make myhello.o
- ❑ \$make myfunc.o
- ❑ \$make hello

```
root@ubuntu:/home/yongza/study/make1# make myhello.o
gcc -c myhello.c
root@ubuntu:/home/yongza/study/make1# make myfunc.o
gcc -c myfunc.c
root@ubuntu:/home/yongza/study/make1# ls
Makefile  myfunc.c  myfunc.o  myhello.o
libmylib.a  myfunc.h  myhello.c  say_linux
root@ubuntu:/home/yongza/study/make1# make hello
gcc -o say_linux myhello.o myfunc.o
```

# make

## ❑ 만들어진 파일들을 청소하는 makefile 만들기

```
hello: myhello.o myfunc.o
    gcc -o say_linux myhello.o myfunc.o
myhello.o: myhello.c
    gcc -c myhello.c
myfunc.o: myfunc.c
    gcc -c myfunc.c
clean :
    rm myfunc.o myhello.o
```

## ■ 실행

### ❑ \$make clean

주의: 명령어 앞에는  
항상 **tab**이 있어야함!

```
root@ubuntu:/home/yongza/study/make1# ls
Makefile  myfunc.c  myfunc.o  myhello.o
libmylib.a myfunc.h  myhello.c  say_linux
root@ubuntu:/home/yongza/study/make1# make clean
rm myfunc.o myhello.o
root@ubuntu:/home/yongza/study/make1# ls
Makefile  libmylib.a  myfunc.c  myfunc.h  myhello.c  say_linux
```

# make

## □ 보다 복잡한 make의 예

```

OFILES = builtins.o main.o run_command.o parser.o handler.o pipe.o
CFILES = builtins.c main.c run_command.c parser.c handler.c pipe.c
CFLAGS = -Wall -g
PARSER = parser
LEX = flex
LFLAGS = -lfl
HEADERS = shell.h
CC = gcc
PROGRAM = shell
all: parser $(PROGRAM)

$(PROGRAM): $(OFILES)
    $(CC) -o $(PROGRAM) $(OFILES) $(LFLAGS) -lreadline

parser: $(PARSER).l
    $(LEX) -o $(PARSER).c $<

$(OFILES): $(HEADERS)
clean:
    rm -f $(OFILES) $(PROGRAM)
```

# 실습 : myls

- 지정된 directory 안의 파일들을 리스팅하는 프로그램

```
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    DIR      *dp;
    struct dirent *dirp;

    if (argc != 2)
    {
        printf("a single argument (the directory name) is required");
        exit(0);
    }
    if ( (dp = opendir(argv[1])) == NULL)
    {
        printf("can't open %s", argv[1]);
        exit(0);
    }

    while ( (dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);
    closedir(dp);
    exit(0);
}
```

# 실습 : myls

## □ 기본 컴파일

- `$gcc -o myls.out myls.c`

## □ 실행

- `./mysls.out <디렉토리>`
- `$.mysls.out .`

```
root@ubuntu:/home/yongza/study# ./mysls.out .
myhello1.c.save
oo
myfunc.o
myfunc.c
mysls.c
libmylib.a
mysls
make1
myhello.out
.
Makefile
myhello1.c
myhello.exe
mysls.out
..
myhello2.exe
myhello.c
myfunc.h
```

# 과제 : myls 확장

## □ 실습과제!!!

- 이 프로그램 중에서 파일들을 리스팅하는 부분을 별도의 c프로그램으로 나누고 라이브러리화 하시오.

```
while ( (dirp = readdir(dp)) != NULL)
    printf("%s\n", dirp->d_name);
closedir(dp);
```

- 라이브러리화된 파일을 이용하여 컴파일 하는 make파일을 만드시오
  - Make를 이용하여 실행파일을 만들고, C소스, 라이브러리, 실행파일을 제외한 오브젝트 파일들은 clean할 수 있게 할 것.
- gdb를 이용하여 debugging 하시오(별도 개인 실습)