

리눅스 시스템 프로그래밍

참고 자료



고강태

010-8269-3535

james@thinkbee.kr



<https://www.linkedin.com/in/thinkbeekr/>



<http://www.facebook.com/gangtai.goh>

GNU Compiler Collection



GCC는 **GNU Compiler Collection**

1999년부터 "GNU Compiler Collection"을 의미한다. 따라서 C 언어뿐만 아니라 C++, 오브젝티브 C, 포트란, 자바 등의 컴파일러를 포함하는 포괄적 의미를 가진다.

1987년 리처드 스톨만이 FSF (Free Software Foundation)에서 공개용 컴파일러로 제작, 이후 다양한 플랫폼, 다양한 언어지원하고 있다.

- 초창기 C, C++ 컴파일러, 어셈블러, 로더 역할로 GNU C Compiler로 불리웠다.
- 이후 여러 플랫폼/언어를 지원하며 GNU Compiler Collection으로 불리고 있다.

GCC Tools

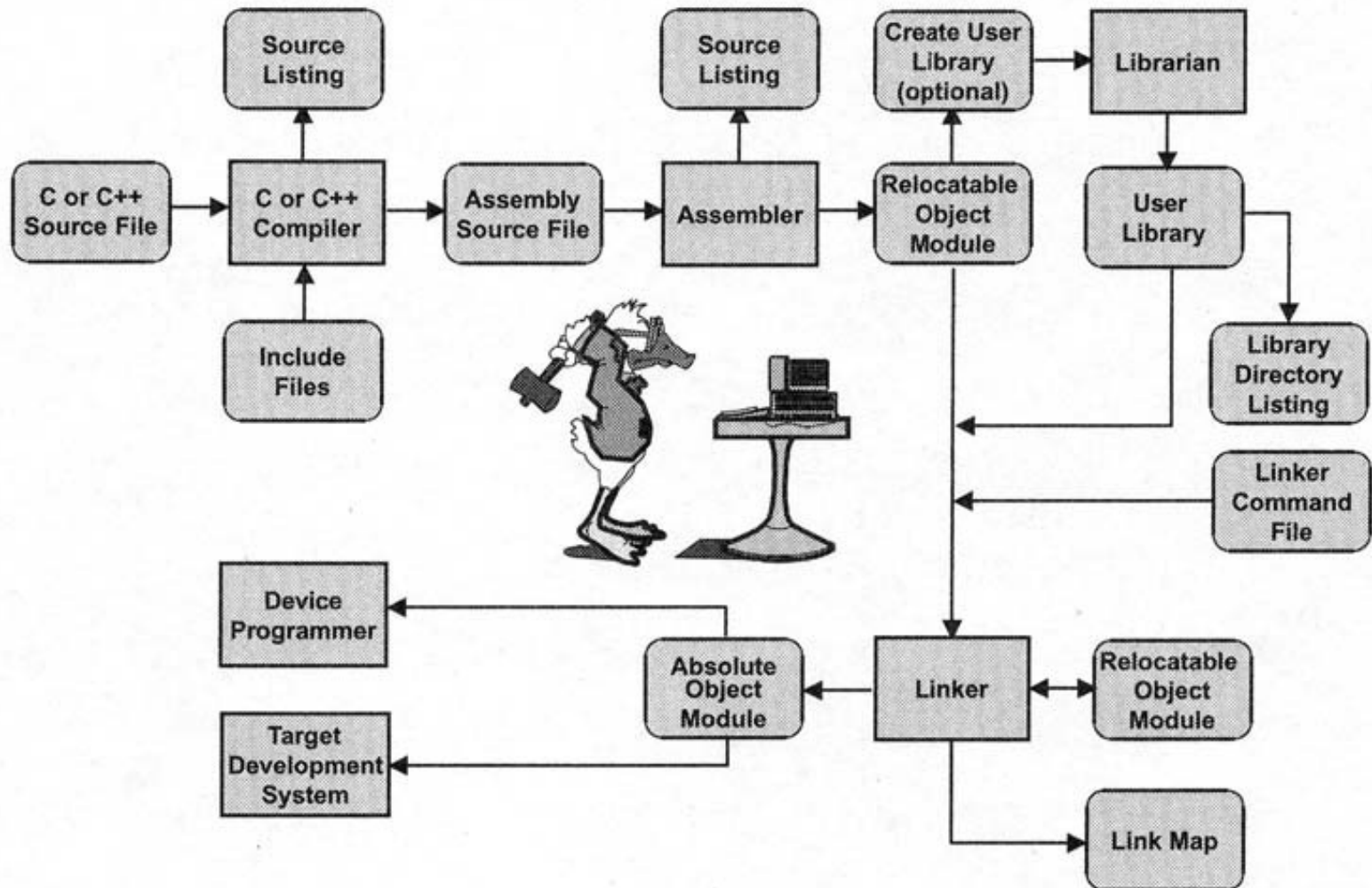
GNU Tool 은 다음을 기본으로 구성하고 있다:

- GNU gcc compilers for C, C++
- GNU binary utilities(assembler, linker various object file utilities)
- GNU C Library
- GNU C header

이들을 Toolchain이라고 하며 보통 툴 체인은 다음과 같은 사항으로 구성되어 있다.

- binutils-arm-2.11.2 : 유틸리티
- gcc-arm-2.95.3 : 컴파일러
- glibc-arm-2.2.3 : 라이브러리

GNU Tool 관계



binutils

binary tool 모듬로 ld, as 라는 CPU native binary 생성 도구
<https://www.gnu.org/software/binutils/>

ld	the GNU linker.
as	the GNU assembler.
addr2line	Converts addresses into filenames and line numbers.
ar	A utility for creating, modifying and extracting from archives.
c++filt	- Filter to demangle encoded C++ symbols.
dlltool	- Creates files for building and using DLLs.
gold	- A new, faster, ELF only linker, still in beta test.
gprof	- Displays profiling information.
nlmconv	- Converts object code into an NLM.
nm	- Lists symbols from object files.
objcopy	- Copies and translates object files.
objdump	- Displays information from object files.
ranlib	- Generates an index to the contents of an archive.
readelf	- Displays information from any ELF format object file.
size	- Lists the section sizes of an object or archive file.
strings	- Lists printable strings from files.
strip	- Discards symbols.
windmc	- A Windows compatible message compiler.
windres	- A compiler for Windows resource files.

GNU Tool 설치

Ubuntu 등 패키지에 기본 개발 환경을 포함한 패키지
- <https://packages.ubuntu.com/bionic/build-essential>

● depends ◆ recommends ■ suggests • enhances

- **dpkg-dev** ($\geq 1.17.11$)
Debian package development tools
- **g++** ($\geq 4:7.2$)
GNU C++ compiler
- **gcc** ($\geq 4:7.2$)
GNU C compiler
- **libc6-dev**
GNU C Library: Development Libraries and Header Files
or **libc-dev**
virtual package provided by **libc6-dev**
- **make**
utility for directing compilation
also a virtual package provided by **make-guile**

```
$ sudo apt install build-essential
```

GNU Tool 설치

라즈베리 파이 배포본 Raspbian에는 기본 **gcc** 개발 환경이 포함되어 있다. 기본적으로 다음 Toolset이 설치되어 있다.

```
$ sudo apt install gcc make
```


컴파일과 실행

다음 hello.c 파일을 컴파일 한다.

```
#include <stdio.h>
int main()
{
    printf ("Hello Linux\n");
    return 0;
}
```

컴파일:

```
~$ gcc hello.c
```

```
~$ ./a.out
```

결과 파일을 -o 옵션으로 지정 (없으면 a.out 로 생성)

```
~$ gcc -o hello hello.c
```

gcc: 기본적인 컴파일 방법

여러 소스 컴파일 과정을 한번에 처리하기

```
$ gcc -o hello hello.c hello_world.c
```

외부 Library 사용하기:

```
$ gcc -o mysqrt mysqrt.c /usr/lib/libm.a
```

혹은

```
$ gcc -o mymath mymath.c -lm
```

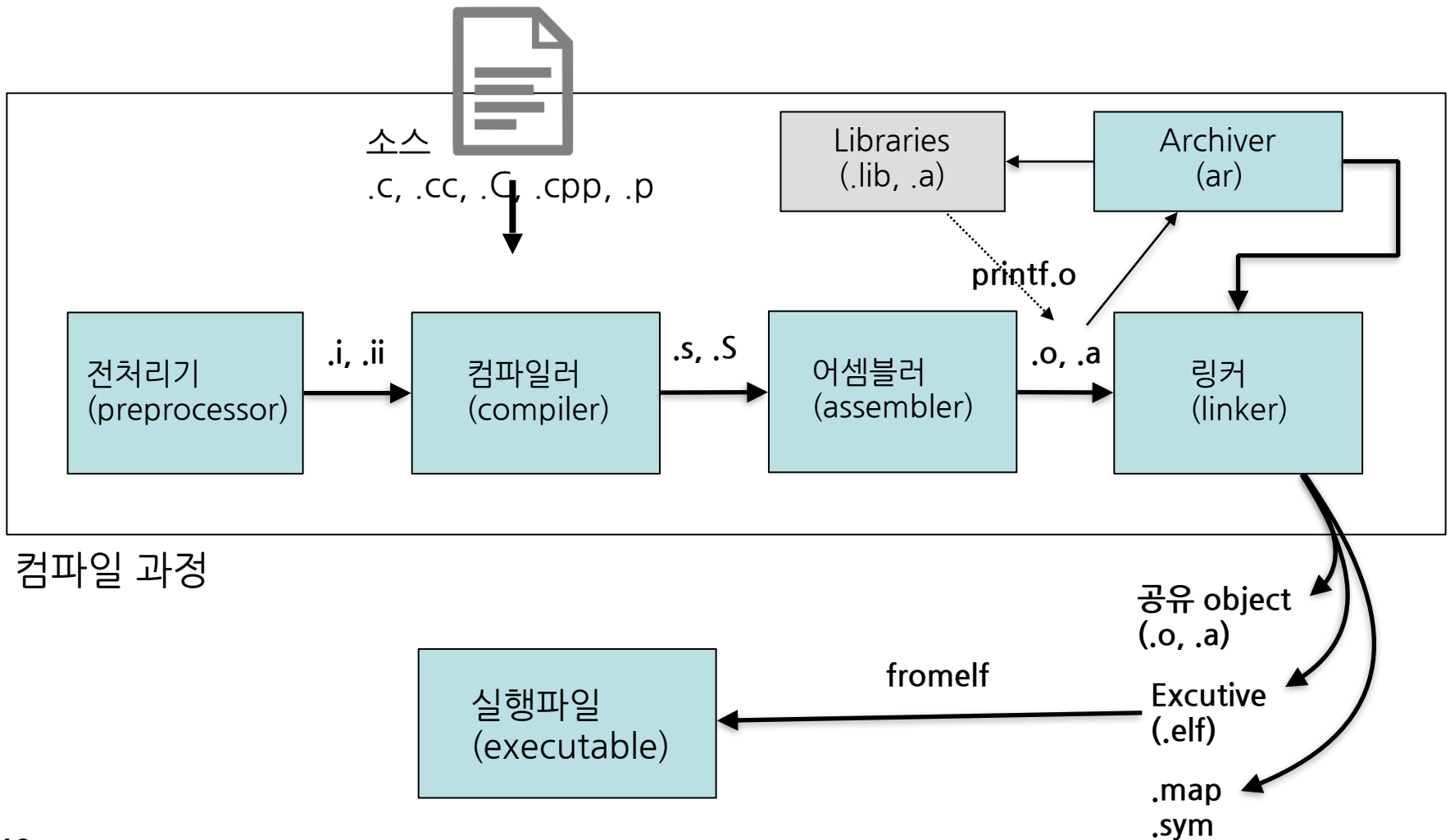


Compile

컴파일러의 컴파일 단계

1. 어휘 분석 (Lexical Analysis)
2. 구문 분석 (Syntax Analysis)
3. 의미 분석 (Semantic Analysis)
4. 중간 코드 생성 (Intermediate Code Generation)
5. 코드 최적화 (Code Optimization)
6. 목적 코드 생성 (Code Generation)

소스 코드 컴파일 과정



Compile 과정

소스를 컴파일 하면

```
gcc -o hello.exe hello.c
```

gcc 컴파일은 전처리, 소스, 어셈블 등 중간 단계 파일을 생성한 후에 최종 linking 과정에서 실행 파일을 생성한다.

Preprocessing

- 확장자 .c 인 소스에 대해 전처리 후 .i 파일이 생성

C source compile

- 전처리 파일을 컴파일해서 .s 어셈블러로 변환

Assembly compile

- 어셈블러로 2진수, 기계어로 형성된 .o 파일 생성

Link

- 실제 실행 파일 (.exe 등)

컴파일러 추론

gcc 컴파일러는 파일 확장자에 따라서 컴파일러를 추론해서 중간 단계 파일을 생성해 Linker 로 전달.

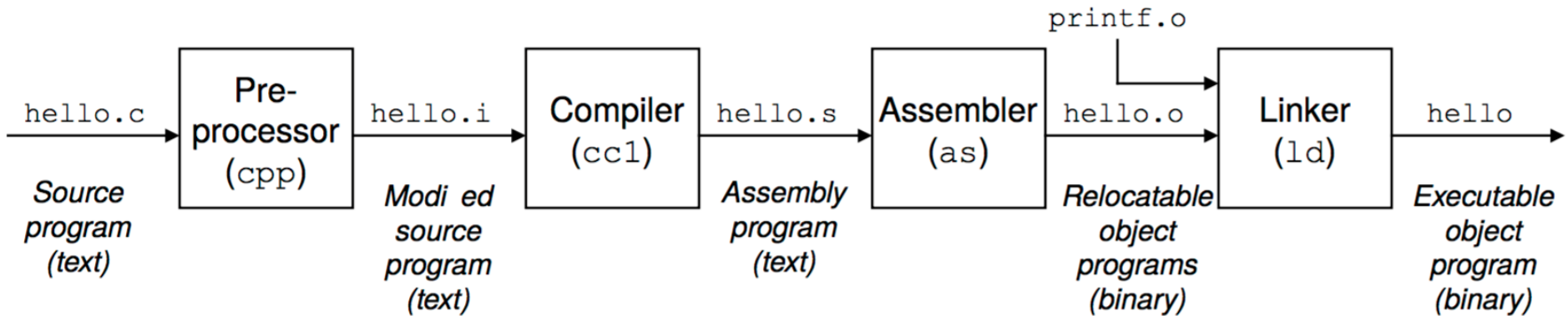
예) 대표적인 확장자 .c 인 경우는 gcc로 전처리, 컴파일, 어셈블, 링크 과정을 거쳐야 Elf 실행 파일이 완성된다.

확장자	종류	처리 방법
.c	C 소스 파일	gcc 전처리, 컴파일, 어셈블, 링크
.c .cc	C++ 소스 파일	g++ 전처리, 컴파일, 어셈블, 링크
.i	전처리된 c 파일	gcc 컴파일, 어셈블, 링크
.ii	전처리된 c++ 파일	g++ 컴파일, 어셈블, 링크
.s	어셈블리어 소스	as 전처리, 어셈블, 링크
.S	어셈블리어 소스	as 전처리, 어셈블, 링크
.o	오브젝트 파일/모듈	ld 링킹
.a .so	오브젝트 라이브러리 파일	ld 링킹

중간단계 파일

컴파일 과정에서 생산되는 중간단계 파일을 컴파일 플래그를 주고 생성할 수 있다.

```
gcc flag -save-temps
```



중간단계 파일은

- 전처리, 디버깅, 컴파일 결과 확인

중간단계 파일 생성

전처리 단계: -E 옵션으로 생성되며 표준 출력으로 내보냄

```
$gcc -E -o hello.i hello.c
```

어셈블리 파일 단계

```
$gcc -S -o hello.s hello.c
```

오브젝트 파일 단계

```
$gcc -c -o hello.o hello.c
```

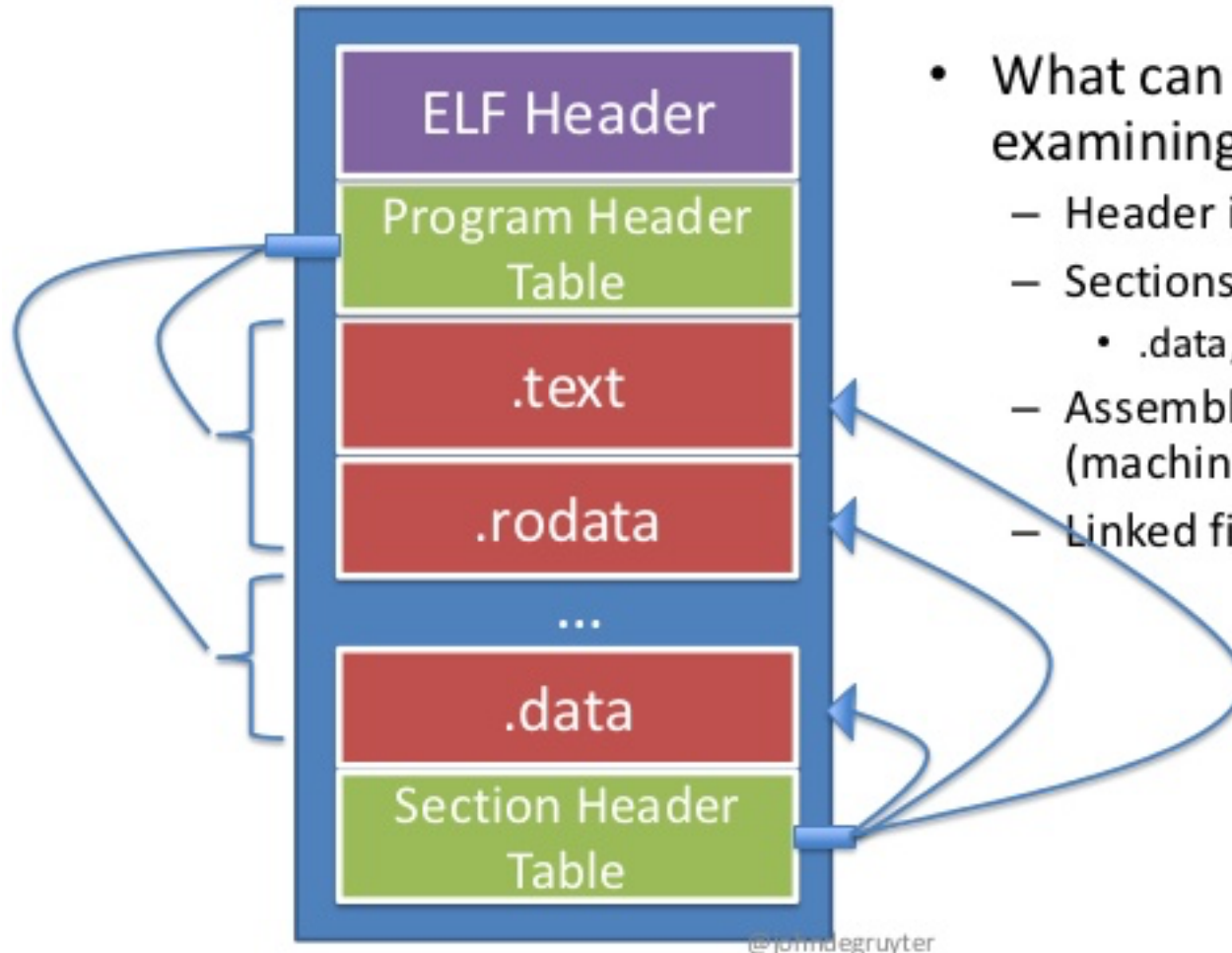
컴파일시 중간단계 파일 저장

```
$gcc --save-temps -o hello hello.c
```

컴파일시 중간단계 화면 출력과 저장

```
$gcc -v --save-temps -o hello hello.c
```

ELF File Structure



- What can we see by examining the file?
 - Header information
 - Sections info and content
 - .data, .rodata, .text
 - Assembly instructions (machine code)
 - Linked files

Tools:
objdump
readelf
ldd
strings

정리: 소스 코드 컴파일 과정

전처리(preprocessing)

- #define, #include, #if 와 같은 전처리 지시자 해석
- 전처리 작업을 위한 cpp 프로그램을 호출

컴파일(compile)

- 고급 언어 소스 프로그램을 입력 받아서 어셈블리 파일을 만듦(.s)
- 일반적으로 어셈블리 파일은 저장하지 않고 바로 어셈블러를 호출함
- 여기서 컴파일은 좁은 의미의 컴파일이며 넓은 의미의 컴파일은 모든 과정을 포함함

어셈블(assembly)

- 어셈블리 파일을 입력 받아서 오브젝트 파일을 만듦(.o)
- 어셈블을 위한 gas 프로그램을 호출함
- 어셈블러는 플랫폼(CPU+OS)마다 다르며 해당 전용 어셈블러 호출 가능

링크(linking)

- 오브젝트 파일을 엮어서 실행 파일을 만듦(.o)
- 라이브러리 함수도 이 단계에서 사용함
- 링크를 위한 ld 프로그램을 호출

첫번째 컴파일

```
#include <stdio.h>

main()
{
    printf("hello world!\n") ;
}
```

```
$ gcc -o hello hello.c
hello.c: In function `main':
hello.c:6: parse error before `}'
$
```

컴파일 시 오류 발생 및 해결

소스 코드를 컴파일할 때 오류가 있을 경우 gcc는 오류 메시지를 출력한다.

오류가 발생하면 소스 코드를 수정하고 다시 컴파일한다.

```
#include <stdio.h>

main()
{
    printf("hello world!\n")
}
```

```
$ gcc -o hello hello.c
hello.c: In function `main':
hello.c:6: parse error before `}'
$
```

문장이 ‘;’으로 종결되지 않았다.

컴파일을 한 결과 소스 코드의 6번 라인에서 오류가 생겼다.

여러 파일 컴파일

두 개 이상의 소스로 하나의 실행 파일 만들기

one.c	two.c
<pre>#include <stdio.h> void printmsg(void); main() { printmsg(); }</pre>	<pre>#include <stdio.h> void printmsg(void) { printf("hello world!\n"); }</pre>

`$ gcc -o three one.c two.c`

출력 파일의 이름은 three이다.

소스 코드 파일은 2개로 각각 one.c와 two.c다.



컴파일러 옵션

gcc 옵션

옵 션	의 미
-v	gcc 버전 정보 출력
-E	전처리를 실행하고 컴파일을 하지 않는다.
-c	소스파일을 오브젝트 파일로만 컴파일하고 링크하는 과정을 생략한다. 즉, 전처리, 컴파일, 어셈블링만 하고, 실행파일을 생성하지 않는다. 컴파일한 결과는 source.c에 대하여 source.o 이다.
-o	바이너리 형식의 출력 파일 이름을 지정하는데, 지정하지 않을 시 a.out 이라고 기본이름 생성
-I	헤더 파일을 검색하는 디렉토리 목록을 추가한다.
-L	라이브러리 파일을 검색하는 디렉토리 목록을 추가한다.
-l	라이브러리 파일을 컴파일 시 링크한다.
-C	전처리기에서 주석을 지우지 않음(디버깅 시 필요)
-DFOO=RAR	명령라인에서 BAR 값을 가지는 FOO 라는 선행 처리기 매크로를 정의한다.
-static	정적 라이브러리에 링크한다.
-shared	가능한 한 공유 라이브러리와 링크. 공유 라이브러리가 없는 경우에만 정적 라이브러리와 링크(기본값)

gcc 옵션(계속)

옵 션	의 미
-s	실행파일을 만들 때 symbolic table을 제거한다.
-O	프로그래밍 최적화 수준을 지정해준다. 예) -O0, -O1, -O2 실행 속도 최적화, 0수준은 전혀 최적화를 안함을 의미한다. 숫자가 클수록 고도화된 실행 속도 최적화 를 수행한다. -Os,이것은 실행파일 크기 최적화를 수행한다.
-S	전처리, 컴파일만 하고 어셈블링 하지는 않는다. 처리 결과는 source.c 에 대하여 source.s 이다.
-Wall	모호한 코딩에 대하여 자세한 경고 메시지
-W	합법적이지만 다소 모호한 코딩에 대하여 부가적인 경고 메시지 출력
-Werror	Warning도 에러로 간주한다.
-U	#define 으로 미리 지정했거나, -D 명령으로 이전에 지정한 정의를 없앤다.
-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 지원한다.
-MM	make 호환의 의존성 목록을 출력한다.
-V	컴파일의 각 단계에서 사용되는 명령을 보여준다.

gcc 옵션

최적화 옵션:

- 프로그램의 수행속도를 컴파일러가 최적화함
- gcc는 최적화와 디버깅 옵션을 동시 사용하도록 허용

옵 션	의 미
-Olevel	Level로 프로그래밍 최적화 수준을 지정해준다. 예) -O0, -O1, -O2 숫자가 클수록 고도화된 실행 속도 최적화를 수행한다.
-O0	최적화 안함(기본값). 정확한 동작/컴파일 시간 적음

디버깅 옵션:

옵 션	의 미
-g	바이너리 파일에 표준 디버깅 정보(심볼 테이블)를 포함한다.
-ggdb	바이너리 파일에 GNU 디버거인 gdb 만이 이해할 수 있는 많은 디버깅 정보를 포함시킨다.
-p	prof에서 프로파일링 할 수 있는 프로그램 생성
-pg	gprof에서 프로파일링 할 수 있는 프로그램 생성

gcc 옵션

부동소수 옵션:

옵 션	의 미
ffast-math	부동 소수점 연산에 대하여 최적화를 수행
finline-functions	단순한 함수의 경우 inline 함수로 변경하여 사용함
fno-inline	모든 inline을 금지
funroll-loops	고정된 반복 횟수의 반복문을 모두 전개함

어셈블러와 링커:

옵 션	의 미
-Wa,option-list	어셈블러에게 옵션 넘겨주기: - Option-list에는 여러 개의 옵션이 콤마(,)로 분리되어 들어감
-Wl,option-list	링커에 옵션 넘겨주기 - Option-list에는 여러 개의 옵션이 콤마(,)로 분리되어 들어감
nostartfiles	: 링크할 때 표준 시스템 구동 파일을 사용 안함 크로스 컴파일 시, 임베디드 프로세서에 사용됨
nostdlib	링크할 때 표준 라이브러리와 구동 파일을 사용 안함 위와 마찬가지로의 경우 타겟 시스템 라이브러리 제공

gcc 옵션: -E

전처리까지만 실행하고 결과를 화면에 출력한다.

전처리 결과를 화면에 뿌리므로 파일에 저장하고 싶으면 출력방향을 파일로 바꿔야 한다.

○ Syntax

gcc **-E** 소스파일이름

예:

```
$gcc -E test.c > test.preprocessed
```

gcc 옵션: -o

-o 옵션

생성되는 출력 파일 이름을 지정한다.

Syntax

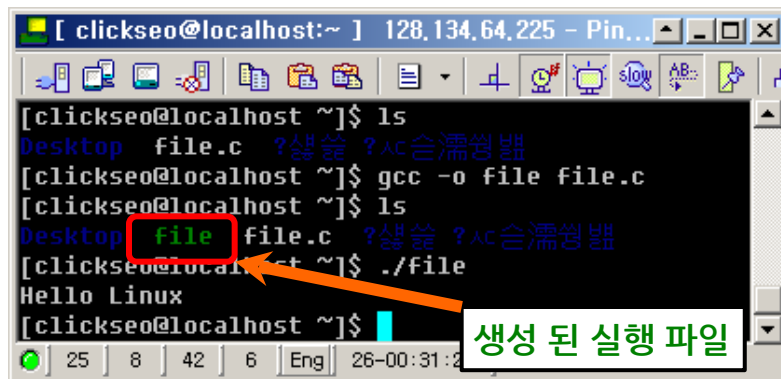
`gcc -o` 출력파일이름 소스파일이름

`gcc` 소스파일이름 `-o` 출력파일이름

(출력파일과 소스파일의 순서는 바뀌어도 상관없다.)

```
$ gcc -o file file.c
```

file.c 소스파일에 file이라는 출력파일 이름을 지정해 주어서 a.out이라는 기본 파일을 생성하지 않는다.



```
[clickseo@localhost:~ ] 128.134.64.225 - Pin...  
[clickseo@localhost ~]$ ls  
Desktop file.c ?생출 ?AC승뽀뽀뽀뽀  
[clickseo@localhost ~]$ gcc -o file file.c  
[clickseo@localhost ~]$ ls  
Desktop file file.c ?생출 ?AC승뽀뽀뽀뽀  
[clickseo@localhost ~]$ ./file  
Hello Linux  
[clickseo@localhost ~]$
```

An orange arrow points from the text '생성된 실행 파일' (Generated executable file) to the 'file' entry in the directory listing.

gcc 옵션: -c

-c 옵션

소스를 오브젝트 파일로만 컴파일하고 링크하는 과정을 생략

Syntax

`gcc -c` 소스파일이름

오브젝트
파일을
이용해
생성 된
실행 파일

```
[clickseo@localhost:~] 128.134.64.225 - PineTerm v2.0.6
[clickseo@localhost ~]$ ls
Desktop file.c
[clickseo@localhost ~]$ gcc -c file.c
[clickseo@localhost ~]$ ls
Desktop file.c file.o
[clickseo@localhost ~]$ gcc file.o -o file
[clickseo@localhost ~]$ ls
Desktop file file.o
[clickseo@localhost ~]$ gcc file.o
[clickseo@localhost ~]$ ls
Desktop a.out file file.c file.o
[clickseo@localhost ~]$
```

생성 된 오브젝트 파일

오브젝트 파일을 이용해 file 이라는 실행 파일 생성

gcc 옵션: -c

-c 옵션 (cont'd)

분리 컴파일

여러 파일로 분리 작성된 하나의 프로그램을 컴파일

[예제 1] main.c

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

```
[clickseo@comlab ~]$ gcc main.c hi.c -o test
```

```
[clickseo@comlab ~]$ gcc -c main.c
```

```
[clickseo@comlab ~]$ gcc -c hi.c
```

```
[clickseo@comlab ~]$ gcc main.o hi.o -o test
```

[예제 2] hi.c

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

```
[clickseo@localhost ~] 128.134.64.225 - PineTerm v2.0.6  
[clickseo@localhost ~]$ ls  
Desktop file.c hi.c main.c  
[clickseo@localhost ~]$ gcc -c main.c  
[clickseo@localhost ~]$ gcc -c hi.c  
[clickseo@localhost ~]$ ls  
Desktop file.c hi.c main.c test  
[clickseo@localhost ~]$ ./test  
Linux World  
[clickseo@localhost ~]$
```

```
[clickseo@localhost ~] 128.134.64.225 - PineTerm v2.0.6  
[clickseo@localhost ~]$ ls  
Desktop file.c hi.c main.c  
[clickseo@localhost ~]$ gcc -c main.c  
[clickseo@localhost ~]$ gcc -c hi.c  
[clickseo@localhost ~]$ ls  
Desktop file.c hi.c hi.o main.c main.o  
[clickseo@localhost ~]$ gcc main.o hi.o -o test  
[clickseo@localhost ~]$ ls  
Desktop file.c hi.c hi.o main.c main.o test  
[clickseo@localhost ~]$ ./test  
Linux World  
[clickseo@localhost ~]$
```

gcc 옵션: -I

-I 옵션

표준 디렉토리가 아닌 위치에 있는 헤더 파일의 디렉토리를 지정한다.

Syntax

gcc 소스파일이름 -I 디렉토리이름

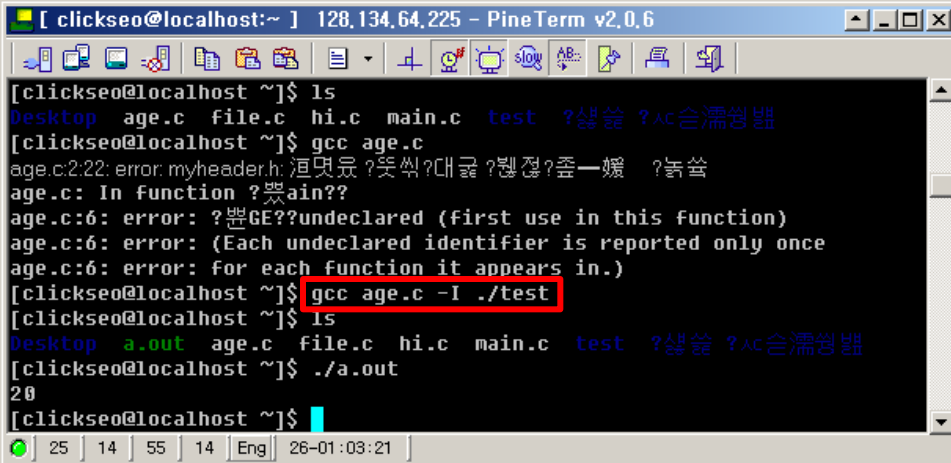
[예제 1] age.c

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

[예제 2] age_header.h

```
#define AGE 20
```

[clickseo@comlab /]\$ gcc age.c -I 헤더파일이 있는 디렉토리 경로



```
[clickseo@localhost ~] 128.134.64.225 - PineTerm v2.0.6
[clickseo@localhost ~]$ ls
Desktop age.c file.c hi.c main.c test ?생물 ?ac습물생물
[clickseo@localhost ~]$ gcc age.c
age.c:2:22: error: myheader.h: 洵뭇윳 ?똥썩?대굴 ?뵘뵘?줄-媛 ?뵘썩
age.c: In function ?뵘ain??
age.c:6: error: ?뵘GE??undeclared (first use in this function)
age.c:6: error: (Each undeclared identifier is reported only once
age.c:6: error: for each function it appears in.)
[clickseo@localhost ~]$ gcc age.c -I ./test
[clickseo@localhost ~]$ ls
Desktop a.out age.c file.c hi.c main.c test ?생물 ?ac습물생물
[clickseo@localhost ~]$ ./a.out
20
[clickseo@localhost ~]$
```


어셈블러(Assembler)

어셈블리 프로그램에서 오브젝트 모듈 생성하고, 각 플랫폼마다 별도의 GNU 어셈블러 존재

어셈블러 실행

\$ as list-of-options list-of-source-files

assemble 옵션

- ah : C 프로그램의 리스트를 생성함
- al: : 어셈블리 언어 코드의 리스트를 생성함
- as : 심볼 테이블에 대한 리스트를 생성함

링커(linker)

ld 링커

오브젝트 모듈과 라이브러리들을 결합하여 실행 파일 생성
외부 변수, 외부 함수, 라이브러리에 대한 참조 위치를 찾아내어 완전한 실행 프로그램 생성

일반적으로 ld를 직접 수행하지 않고 gcc로 수행함
모든 라이브러리의 위치를 일일이 적어 줄 필요 없음
기본 라이브러리를 포함 시켜 줌

ld 실행

`$ ld list-of-options list-of-files-and-libraries`

라이브러리는 `-llib-name` 형태로 사용, 파일과 라이브러리 순서 중요
라이브러리의 경우 외부 참조에 의하여 필요한 함수만 추출함

`$ ld prog1.o -lm prog2.o ==>` prog2는 math 함수를 사용하지 않음
순서와 관계없이 라이브러리에서 모듈을 찾기 위하여 색인을 생성함
`$ ld prog1.o -lat -lfo ==>` fo 라이브러리에서 at 함수 사용 시 에러
ranlib를 사용하여 색인을 만들어 주면 에러 발생 안함

링커(linker)

ld 실행 파일 만들기

실행 파일 생성시 시작 위치는 첫째 파일의 시작 부분임

C 프로그램의 시작점과 같지는 않음

OS에서 프로그램 시작 전에 표준 런타임 초기화 루틴을 실행
플랫폼마다 제공되며, 보통 유닉스 계열에서는 /lib/crt0.o

gcc와 ld

다음의 두 표현은 동일

```
$ gcc test.o
```

```
$ ld -dc -dp -e start -X -o a.out /usr/lib/crt0.o expo.o -lc
```

링커(linker)

ld 옵션들

```
-o name : 실행 파일 이름 지정
-lname  : 표준 라이브러리 링크 (/lib /usr/lib)
-Ldir   : 표준 라이브러리 디렉토리 추가
-s       : 실행 파일에 심볼 테이블 제거 (ref. strip 명령)
-x       : 출력 파일에 로컬 심볼 제거
-n       : 텍스트 영역을 읽기 전용으로 만들
-r       : 추후에도 링크 할 수 있도록 오브젝트 파일 만들
-e name  : 실행 파일의 시작위치를 name 심볼로 사용
-M       : 전역 심볼의 값이 어느 함수에 위치하는지 보여주는 로드맵 작성
-b format : 오브젝트 파일을 주어진 형식으로 읽어 들임
-oformat format: 주어진 형식의 오브젝트 파일을 생성
```