HALLYM UNIVERSITY

Operating Systems LAB 2

Wonpyo Kim

skykwp@gmail.com

Outline

- Substance
  - VIM
  - GCC

VIM ?

- Vim 에디터는 vi 에디터의 진보된 형태로, vi 에디터는 유닉스의 창시자였던 켄 톰슨(Ken Thomsom)의 제자이자 동료인 빌 조이가 1976년 라인 에디터를 개량해 만든 것이며, 텍스트 기반의 CUI 환경에서 주로 사용된다

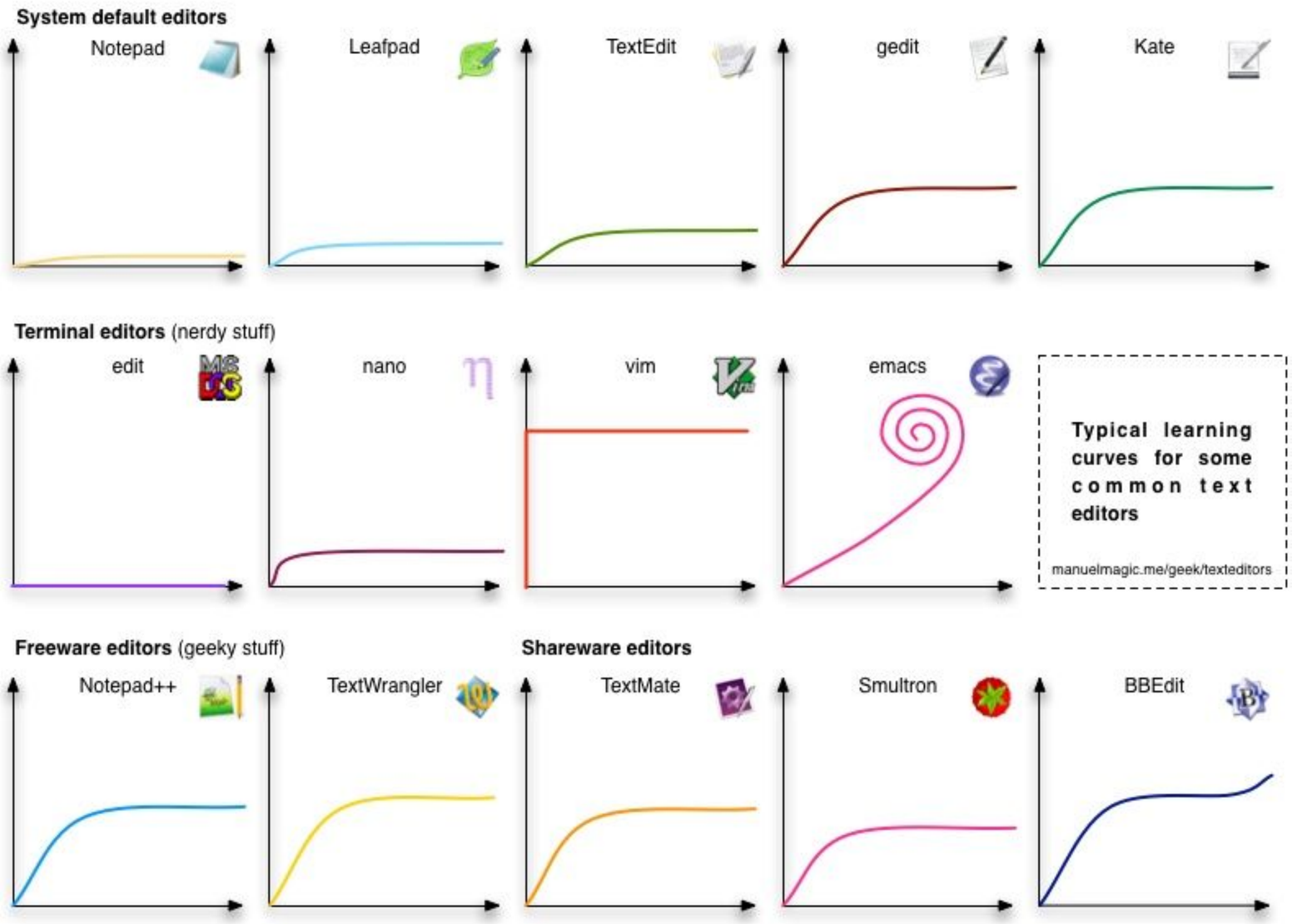
- Vim은 GUI 환경의 에디터들과 달리 실행속도가 수십배는 빠르고 사용하는 리소스도 현저히 적다

- 국제화 규격에 맞춰 유니코드나 다른 언어권의 문자도 지원한다

- Vim = vi 에디터 이지만, vim은 vi improved 의 줄임말로 vi의 기본기능을 계승하면서 편의기능이 추가된 것이다

- Vim은 키보드만 사용해서 모든 작업을 처리할 수 있다는 것 또한 매우 큰 장점이다

Why VIM ?


System default editors
Notepad
Leafpad
TextEdit
gedit
Kate
Terminal editors (nerdy stuff)
edit
nano
vim
emacs
Typical learning curves for some common text editors
manuelmagic.me/geek/texteditors
Freeware editors (geeky stuff)
Notepad++
TextWrangler
Shareware editors
TextMate
Smultron
BBEdit

VIM Mode

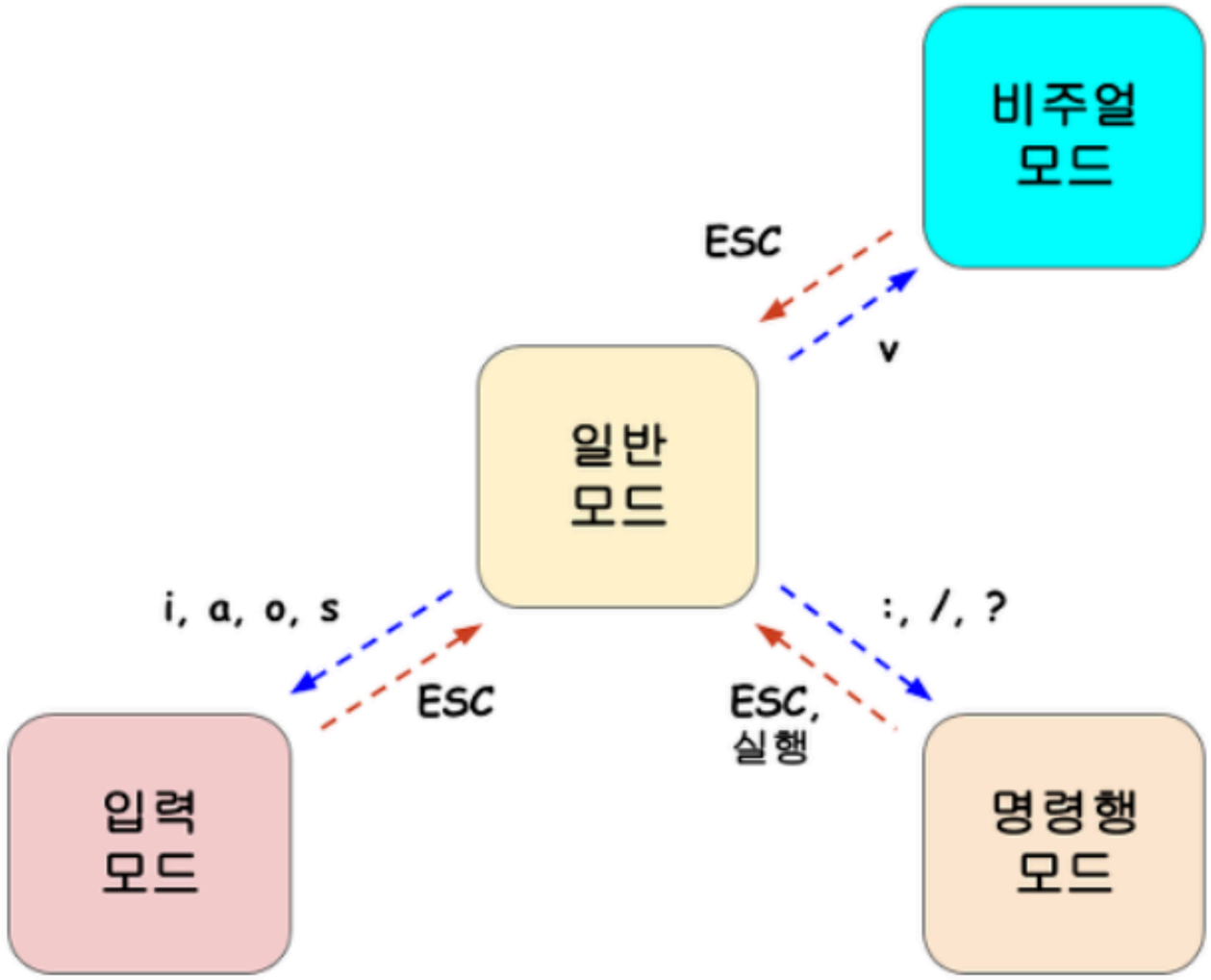
- 모드의 종류
  - 일반모드
    - 키 입력을 통해 vim에 명령을 내린다. 즉, 커서를 이동하거나 삭제, 복사, 붙여넣기 등의 작업을 수행할 수 있다.
    - vim이 처음 실행되면 이 모드가 된다.

  - 입력모드
    - 실제로 문서를 편집하기 위한 모드이다. 다른 에디터들의 입력창에 커서를 놓는 것과 같은 이치이다.

- 명령라인 모드
  - 명령입력을 통해 다른 일을 수행하는 모드이다. 설정을 바꾸거나, 다른 파일을 열거나, 저장을 할 수 있다.

- 비주얼 모드
  - vim에서는 명령을 수행할 때 범위(라인)를 지정할 수 있다.
  - 문서 전체를 지정하거나, 특정 행부터 특정 행까지 혹은 N개의 행 등의 여러가지 방법으로 범위를 지정할 수 있다
  - 일반 에디터에서 마우스로 드래그하는 것과 같다

VIM Mode



VIM Mode

- 실습
  - 입력모드

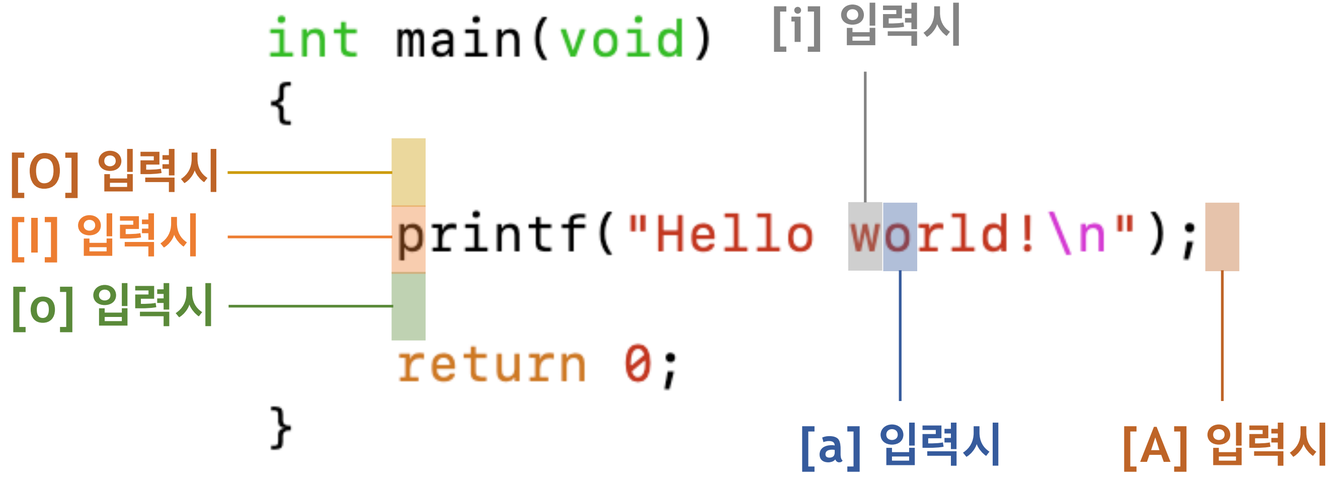
```c
int main(void)
{
    printf("Hello world!\n");

    return 0;
}
```

현재 커서의 위치가 "w"에 있다고 가정해보자

VIM Mode

- 실습
  - [i], [I], [o], [O], [a], [A] 입력 시 커서의 위치는 다음과 같다. 한 개씩 해보면서 커서의 위치이동을 살펴보아라



VIM Mode

- 입력모드
  - **i** : 현재 커서의 위치에서 입력을 시작한다
  - **I** : 현재 커서가 있는 행의 맨 앞에서 입력을 시작
  - **a** : 현재 커서의 바로 다음 문자부터 입력을 시작
  - **A** : 현재 커서가 있는 행의 맨 뒤에서 입력을 시작
  - **o** : 현재 커서가 있는 행의 다음 행부터 입력을 시작
  - **O** : 현재 커서가 있는 행의 이전 행부터 입력을 시작

  - **s** : 현재 커서가 위치한 글자를 지우고 입력을 시작
  - **S** : 현재 커서가 위치한 행을 지우고 입력을 시작

  - **R** : 현재 커서가 있는 곳에서 [수정모드]로 작업을 한다

VIM Mode

- 명령모드
  - 명령모드는 일반모드에서 [:] 키(**Shift + ;**)를 입력하면 된다
  - 그러면 화면 하단에 : 문자가 표시되고 입력을 받는 프롬프트가 표시된다
  - **[/]**, **[?]** 키를 입력해도 명령모드가 되는데, 이는 **정방향 [/]**, **역방향 [?]** 검색을 할 때 사용한다. 커서의 위치에 의존하지 않고 해당 파일 내용을 검색한다

- 비주얼모드
  - 비주얼모드는 일반모드에서 **[v]**, **[V]** 키를 통해 접근한다
  - 비주얼모드에서는 블록을 지정해줄 수 있다 (드래그 앤 드롭). 물론 키보드 방향키로 지정한다

- 모든 모드는 **[일반모드]**로 돌아간 후에 전환이 가능하다. 즉, 명령모드에서 바로 입력모드로는 전환할 수 없다

VIM Mode

- 커서이동
  - 커서이동은 **[일반모드]**에서 수행할 수 있으며 다음과 같다

    - **[#]gg** : [#] 번째 행으로 이동한다 (생략시 첫 번째 행)
    - **[#]G** : [#] 번째 행으로 이동한다 (생략시 마지막 행)
    - **:[#]** : [#] 번째 행으로 이동한다

- 실행예
  - **3gg**
  - **5G**
  - **:10**

    - **참고로 일반모드에서 수행하기 때문에 : 명령 외에는 표시되는 것이 없기 때문에 멈춘것으로 오인할 수 있다**

VIM Mode

- 편집기능
  - 문서 및 코딩 작업을 하다보면 특정 행이 여러번 필요한 경우가 있다. 그럴 경우 복사, 붙여넣기, 삭제, 실행취소 등을 수행해야한다. Vim에서는 그러한 기능을 모두 지원한다
  - 다음 명령들 역시 **[일반모드]**에서 수행해야 한다

- 복사
  - **[y]** : 레지스터에 복사
  - **[yy]** or **[:y]** or **[Y]** : 현재 행을 레지스터에 복사
    - tip: 이전 장에서 **[5gg]** 등의 행이동을 응용하면 **[5yy]** 입력으로 5줄의 행을 레지스터에 복사할 수 있다

- 붙여넣기 (레지스터에 내용이 있어야 한다)
  - **[p]** or **[:pu]** : 현재 행에 붙여넣기, 개행 문자가 포함된 경우에는 현재 행의 아래에 추가된다
  - **[P]** : 현재 행의 위쪽에 붙여넣기

VIM Mode

- 삭제(잘라내기)
  - **[x]** : 커서에 위치한 문자 삭제
  - **[dd]** or **[:d]** : 현재 행 삭제
  - **[D]** or **[d$]** : 현재 컬럼 위치에서 현재 행의 끝부분까지 삭제
  - **[J]** : 아래 행을 현재 행의 끝에 붙임 (아래 행의 앞부분 공백은 제거됨)

- 실행취소
  - **[u]** : undo 기능으로 바로 직전명령을 취소할 수 있다
  - **[Ctrl + R]** : redo 기능으로 바로 직전에 취소했던(undo) 명령이 되돌려진다

VIM

- VIM으로 게임을 할 수 있다. 오직 VIM에서 사용하는 키로만 진행할 수 있다.
  - https://vim-adventures.com/

- 줄번호, 탭간격 조정은 개인의 개발환경에 따라 달라질 수 있다. 아래에서 설정하는 방법을 잘 나타내주고 있다.
  - https://dougblack.io/words/a-good-vimrc.html

Outline

- Substance
  - VIM
  - GCC

C and GCC

- 유닉스가 C언어로 대부분 만들어 졌듯이, 리눅스는 gcc라고 해도 과언이 아니다

- gcc는 매우 단순하며, 강력하다. 다른 의미로는 윈도우즈 처럼 통합적인 환경을 제공해주지 않는다. 즉, 윈도우즈에서 보던 것을 원하면 안된다.

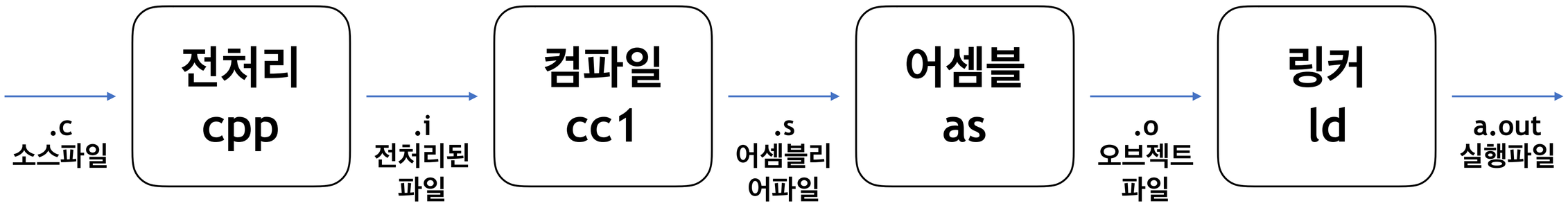
GCC

- 쉘에서 다음을 실행해보자. 자신이 사용하고 있는 gcc의 버전은 알아둘 필요가 있다.

```
kwp@A1409-OSLAB-01:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.3.0-27ubuntu1~18.04' --with-bugurl=file
:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++ --prefix
=/usr --with-gcc-major-version-only --program-suffix=-7 --program-prefix=x86_64-linux-gnu- --enable-shar
ed --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --li
bdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libst
dcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enab
le-libmpx --enable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib --enable-obj
c-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m3
2,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none --without-cuda-driv
er --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.3.0 (Ubuntu 7.3.0-27ubuntu1~18.04)
kwp@A1409-OSLAB-01:~$
```

Usage of GCC

- GCC가 실행시키는 프로그램
  - cpp : 전처리기
  - cc1 : 컴파일러
  - as : 어셈블러
  - ld : 링커



Usage of GCC

- 전처리 단계
  - #include, #define 등의 전처리 부분을 처리
  - .i 확장자를 가진 파일을 생성

- 컴파일 단계
  - 전처리된 파일을 이용하여 어셈블리어로 된 파일을 생성
  - .s 확장자를 가진 파일을 생성

- 어셈블 단계
  - 어셈블리어로된 파일을 기계가 이해할 수 있는 기계어로 된 오브젝트 파일을 생성
  - .o 확장자를 가진 파일을 생성

- 링크 단계
  - printf, scanf 와 같은 라이브러리 함수와 오브젝트 파일을 연결하여 실행파일로 생성

Usage of GCC

- -E 옵션
  - 전처리만을 수행한다

- -S 옵션
  - 컴파일만을 수행한다

- -c 옵션
  - 소스 파일을 컴파일과 어셈블을 수행한다

- -o 옵션
  - 바이너리 형식의 출력파일 이름을 지정한다.

Usage of GCC

- -E 옵션만 사용

```
kwp@A1409-OSLAB-01:~/toy/lab2$ cat hello.c

#include <stdio.h>

void main()
{
        printf("hello?\n");
}
```

```
kwp@A1409-OSLAB-01:~/toy/lab2$ gcc -E hello.c
extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
# 840 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 868 "/usr/include/stdio.h" 3 4

# 3 "hello.c" 2


# 4 "hello.c"
void main()
{
 printf("hello?\n");
}
kwp@A1409-OSLAB-01:~/toy/lab2$
```

Usage of GCC

- -E 옵션만 사용
  - 앞서 언급한 .i 파일은 자동으로 생성하지 않는다. 따라서, -o 옵션을 이용한다

```
kwp@A1409-OSLAB-01:~/toy/lab2$ gcc -E hello.c -o hello.i
kwp@A1409-OSLAB-01:~/toy/lab2$ ls
a.out  bar.c  baz  foo.c  hello.c  hello.i  kindergarten  kindergarten.c  kindergarten.o
kwp@A1409-OSLAB-01:~/toy/lab2$
```

  - .i 파일이 생성된 것을 볼 수 있다

Usage of GCC

- -S 옵션만 사용

```
kwp@A1409-OSLAB-01:~/toy/lab2$ gcc -S hello.c
kwp@A1409-OSLAB-01:~/toy/lab2$ cat hello.s
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "hello?"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
        .section        .note.GNU-stack,"",@progbits
kwp@A1409-OSLAB-01:~/toy/lab2$
```

- 컴퓨터 구조 과목에서 배울 수 있는 어셈블리어로 나타나 있는 것을 볼 수 있다
- 생소한 명령들이 있는 까닭은 세대를 거듭하면서 최적화 및 새로운 명령이 추가되었기 때문이다

Usage of GCC

- -c 옵션만 사용

```
[kwp@A1409-OSLAB-01:~/toy/lab2$ gcc -c hello.c
[kwp@A1409-OSLAB-01:~/toy/lab2$ cat hello.o
ELF>?@@
UH??H?=??]?hello?GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0zRx
N                                                    A?C
??      $hello.cmain_GLOBAL_OFFSET_TABLE_puts????????

                                                ???????? .symtab.strtab.shstrtab.rela.text.data.bss
.rodata.comment.note.GNU-stack.rela.eh_frame @0
&SS1S90Z+B?W?R@@
?
[       ?)Xakwp@A1409-OSLAB-01:~/toy/lab2$
kwp@A1409-OSLAB-01:~/toy/lab2$
```

- .o 파일은 기계어로 되어 내용을 확인하기 어렵다

Usage of GCC

- -o 옵션만 사용

```
[kwp@A1409-OSLAB-01:~/toy/lab2$ gcc -o hello hello.c
[kwp@A1409-OSLAB-01:~/toy/lab2$ ./hello
 hello?
[kwp@A1409-OSLAB-01:~/toy/lab2$ _
```

Usage of GCC

- 소스파일이 하나만 있는 경우라면 앞선 방법으로도 가능하다. 하지만, 프로젝트에 따라 수십 내지 수백개의 파일이 존재할 수 있다. 그럴경우 오브젝트 파일을 생성해서 엮어주는 작업이 필요하다.

- main.c 와 print.c 를 다음과 같이 생성한다

```
kwp@A1409-OSLAB-01:~/toy/lab2$ cat main.c
#include <stdio.h>

extern void print();

void main()
{
        print();
}

kwp@A1409-OSLAB-01:~/toy/lab2$ cat print.c
#include <stdio.h>

void print()
{
        printf("This is print() function.\n");
}

kwp@A1409-OSLAB-01:~/toy/lab2$ _
```

Usage of GCC

- 이전처럼 main.c를 컴파일하면 정의되지 않은 참조인 print 함수를 사용했다는 오류가 발생한다

```
kwp@A1409-OSLAB-01:~/toy/lab2$ gcc -o print_test main.c
/tmp/ccz49cTz.o: In function `main':
main.c:(.text+0xa): undefined reference to `print'
collect2: error: ld returned 1 exit status
kwp@A1409-OSLAB-01:~/toy/lab2$
```

- 여러 개를 합치려면 다음과 같이 파일을 명시해야 한다. 명령이 간단한 이유는 gcc에서 자동으로 오브젝트 파일을 생성한 후 엮기 때문이다.

```
kwp@A1409-OSLAB-01:~/toy/lab2$ gcc main.c print.c -o print_test
kwp@A1409-OSLAB-01:~/toy/lab2$ ./print_test
This is print() function.
```

- 정석은 다음과 같다. 이 방법을 권장한다

```
kwp@A1409-OSLAB-01:~/toy/lab2$ gcc -c main.c
kwp@A1409-OSLAB-01:~/toy/lab2$ gcc -c print.c
kwp@A1409-OSLAB-01:~/toy/lab2$ ls
a.out  baz    hello    hello.i  hello.s      kindergarten.c  main.c  print.c  print_test
bar.c  foo.c  hello.c  hello.o  kindergarten  kindergarten.o  main.o  print.o
kwp@A1409-OSLAB-01:~/toy/lab2$
```

Usage of GCC

- 정석을 강조하는 이유는 특정 파일만 수정하고자 할 때, 그 파일만 수정해서 오브젝트 파일로 만들어주면 다른 파일은 건드리지 않아도 된다

- print.c 를 수정하고 print.c 만을 컴파일해서 확인한다

```
kwp@A1409-OSLAB-01:~/toy/lab2$ cat print.c
#include <stdio.h>

void print()
{
        printf("This is print() function.\n");
        printf("Really?\n");
}


kwp@A1409-OSLAB-01:~/toy/lab2$ gcc -c print.c
kwp@A1409-OSLAB-01:~/toy/lab2$ gcc main.o print.o -o print_test
kwp@A1409-OSLAB-01:~/toy/lab2$ ./print_test
This is print() function.
Really?
kwp@A1409-OSLAB-01:~/toy/lab2$
```