1. 개 요

UNIX 환경에서의 프로그래밍은 Windows 환경에서의 그것과는 사뭇 다른 모습을 지닌다. Windows에서는 Visual C++이나 Turbo C, Borland C와 같은 GUI기반의 화려한 통합 개발 환경(IDE, Integrated Development Environment)을 제공하고 있다. 이에 비하여 UNIX 환경의 개발 환경은 초라하기만 하다. 화려한 GUI 기능이 있는 것도 아니며, 윈도우 기반의 프로그래밍에 강점이 있는 것도 아니다.

하지만, UNIX는 어떠한 여타의 운영체제보다도 C와의 인연이 깊다고 할 수 있다. 가장 처음으로 C언어로 작성된 운영체제도 UNIX이며 따라서 C 개발 관련 도구 또한 막강한 기능들을 자랑한다. 물론, UNIX가 본래 GUI 환경으로 개발된 것이 아니어서, 이 부분에서는 취약점을 보이지만 그 외의 부분에서는 UNIX만큼 훌륭한 C언어 개발 도구들을 지닌 운영체제도 없을 것이다.

UNIX에는 앞서 살펴 본 바와 같이 강력한 편집 기능을 자랑하는 vi, 뛰어난 기능과 안정성을 보이는 C언어 컴파일러인 gcc(GNU C Compiler), Windows의 Visual C++보다 더욱 향상된 기능을 제공하는 gdb, 소스 코드의 컴파일을 제어 할 뿐만 아니라 어플리케이션의 설치까지 지원해 주는 make, 여러 명이 동시에 프로그램을 개발할 때 소스 코드를 관리해 주는 cvs 등 유용한 개발 도구들로 꽉 차 있다.

여기에서는 이러한 도구 중에서, C 코드를 컴파일하고 디버깅할 때 사용하는 도구들 즉 gcc, gdb, make에 대하여 살펴보도록 하겠다.

2. 개발 도구 소개

2-1. gcc

2-1-1. 개요

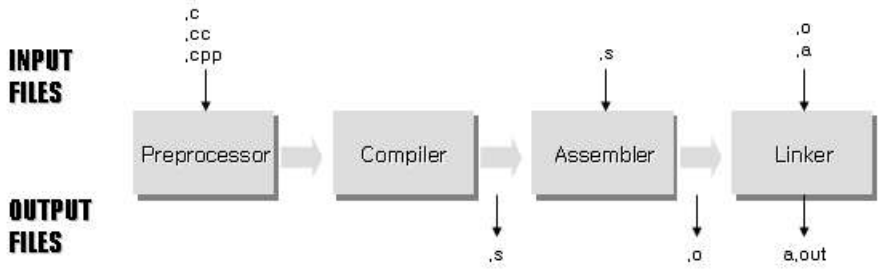

그림 1. 컴파일 과정.

C의 컴파일 과정의 위의 그림과 같은 단계를 거친다. gcc는 이와 같은 컴파일 과정에서 거의 모든 부분을 담당하고 있다. 본래 gcc가 하는 역할은 preprocessor와 compiler의 역할이지만 내부적으로 UNIX의 assembler인 as와 linker인 ld를 호출함으로써, 전 과정을 맡고 있다. 이와 같이 gcc는 UNIX에서 C언어로 프로그램을 개발하는데 있어 가장 중요한 프로그램이라 할 수 있다.

2-1-2. 실행 방법

$ 실행 방법

```
gcc [-c|-S|-E] [-std=standard]
    [-g] [-pg] [-Olevel]
    [-Wwarn...] [-pedantic]
    [-Idir...] [-Ldir...]
    [-Dmacro[=defn]...] [-Umacro]
    [-foption...] [-mmachine-option...]
    [-o outfile] infile...
```

gcc로 컴파일하는 가장 간단한 방법은 gcc 명령어 뒤에 C 소스 파일들을 열거하면 된다. 다음의 예제는 gcc로 컴파일하는 가장 단순한 방법을 나타낸다.

☞ 예제

```
$ ls
main.c
$ gcc main.c
$ ls
a.out   main.c
$
```

기본적인 컴파일 이외에 여러 다른 목적으로 인하여 옵션이 추가되게 된다. 중요한 옵션들은 다음과 같다.

표 1. gcc 옵션.

옵 션	설 명
-o *name*	실행 파일명을 *name*으로 대치한다. 이 옵션이 없는 경우 gcc는 기본적으로 실행파일 명을 a.out으로 한다. 예: gcc -o myapp main.c a.c b.c
-S	컴파일 과정에서 assmble 이전 과정만 수행한다. 실행 결과 해당 C 소스파일의 어셈블코드 인 .S 파일이 생성된다.
-c	컴파일 과정에서 linking 이전 과정만 수행한다. 실행 결과 해당 c 소스파일의 목적파일(object file)인 .o 파일이 생성된다.
-I*dir*	include 파일을 찾을 때 해당 *dir*에서 검색한다. include 파일이 현재 디렉토리에 존재하지 않는 경우 이 옵션을 필요로 한다. 예: gcc -I../include/ main.c a.c b.c
-D*macro*[=*defn*]	gcc에서 macro를 선언한다. 이는 c 소스코드에서 #define *macro* 를 선언해준 것과 같은 효과를 낸다.
-w	컴파일 시 발생하는 모든 경고(warning)메시지를 출력하지 않는다.
-Wall	경고 메시지뿐만 아니라 의문시되는 코딩 습관에 대해서도 경고를 출력한다.
-O	코드를 최적화하여 컴파일한다. 이 옵션을 주면 보통 코드의 수행 속도 향상을 기대할 수 있다. -O0 최적화를 하지 않는다. 기본적으로 설정되는 옵션이다. -O1 컴파일된 코드의 크기를 줄이고 실행 속도를 빠르게 한다. -O0 옵션보다 컴파일되는 속도가 느리다. -O2 -O1 옵션 보다 좀더 최적화되도록 컴파일한다.

-l*name*	프로그램을 lib*name*.a 라이브러리 파일과 링킹한다. 기본적으로 라이브러리 파일은 /lib나 /usr/lib 디렉토리에서 검색한다. 수학 함수 관련 라이브러리(libm.a)를 링킹하여 컴파일하는 예는 다음과 같다. 예: gcc mian.c a.c b.c -lm
-L*dir*	*dir* 디렉토리에서 라이브러리 파일을 찾는다.
-g	디버깅을 위한 정보를 포함하여 컴파일 한다. gdb를 사용하기 위해서는 꼭 이 옵션을 사용하여 컴파일 하여야 한다.

2-2. gdb

2-2-1. 개요

일반적으로 모든 소프트웨어는 크고 작은 결함을 갖게 마련이다. 이런 문제점은 프로그램과 라이브러리가 예상대로 동작하지 않도록 만든다. 문제점의 원인에는 여러 가지가 있을 수 있다. 가령 코드를 입력하는 과정에서 실수를 했을 수도 있고, 또는 설계 시에 설계를 잘못 할 수도 있을 것이다. 버그(bug)란 이러한 원인들로 인하여 발생한 결함들을 지칭하는 말이고, 디버깅(debugging)이란 이러한 버그를 찾아 없애는 작업을 의미한다.

gdb는 디버깅 툴이다. 즉 gdb는 디버깅을 유용하게 할 수 있는 환경을 제공해 준다.

여기에서는 디버깅 방법에 대한 자세한 설명은 생략하고 gdb의 사용 방법에 대하여 제시하고자 한다. 디버깅 방법에 대해서는 뒤의 실습에서 익혀 보도록 하자.

2-2-2. 실행 방법

$ 실행 방법

```
gdb    [-help] [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps] [-tty=dev]
       [-s symfile] [-e prog] [-se prog] [-c core] [-x cmds] [-d dir]
       [prog[core|procID]]
```

gdb는 다음과 같이 간단히 실행시킬 수 있다.

☞ 예제

```
$ gdb qsort2
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux".
(gdb)
```

디버깅할 프로그램을 인자로 주고 실행 시키면 gdb의 copyright와 함께 프롬프트인 (gdb)가 화면에 나타난다.

2-2-3. gdb 명령

- 소스 파일 코드 보기

표 2. Listing.

명 령	설 명
list	실행 부분의 코드를 나열한다. 기본적으로 10줄의 코드를 나열하며, list 명령을 다시 한번 사용하면 그 다음 10줄의 코드가 나열된다. 단축키 : l
list *line1*	라인 번호 *line1* 사이의 10줄의 코드가 나열된다. 즉 (*line1* - 5) ~ (*line1* + 4) 사이의 코드들이 나열된다.
list *line1*, *line2*	*line1*과 *line2* 사이의 코드들이 나열된다.
list *routine-name*	*routine-name*의 루틴 코드들이 나열된다. 특정 함수의 코드를 살펴볼 때 유용한 명령어이다.

- 프로그램 실행

표 3. Executing.

명 령	설 명
run [*args*]	프로그램을 실행한다. *args*가 주어지면 해당 인자를 가지고 프로그램을 실행한다. 실행 도중 CTRL+C 키를 누르면 실행되던 코드 부분을 출력하면서 중단된다. 단축키 : r
set args *argument*	프로그램 실행 시 사용하는 인자를 *argument*로 설정한다. 설정 이후에 run을 실행하면 자동으로 인자가 추가된다.
show args	현재 설정되어 있는 인자들을 출력한다.

- 데이터 출력

표 4. Printing data.

명 령	설 명
whatis *var*	*var* 의 타입(type)을 출력한다.
print *var*	*var* 의 값을 출력한다. 값이 출력될 때에는 출력될 때에는 $표시와 함께 숫자가 출력된다. 이것은 value history라 불리며, 나중에 $num 의 형태로 그 값을 재사용할 수 있다. 예: (gdb) print p $1 = (int *) 0xf8000000
print *routine*(*args*)	인자와 함께 함수를 실행하여 그 결과 값을 출력한다. 단, 전역 변수의 값에 영향을 미칠 수도 있으니 조심해서 사용해야 한다.
print $*num*	value history를 출력한다. 보통은 value history와 함께 연산을 하여 결과 값을 출력 할 때 많이 쓰인다. 예: (gdb) print $1-1 $2 = (int *) 0xf7fffffc
print *base*@*length*	*base*로부터 *length* 만큼의 array를 출력한다. 보통은 동적으로 할당받은 array를 살필 때 사용한다. 예: (gdb) print h@10 $3 = {-1, 349, 0, 0, 0, 536903697, 32831, 1, 0}

▪ Breakpoints

표 5. Breakpoints.

명 령	설 명
break *line-number*	프로그램이 실행되는 코드가 *line-number* 의 코드이면 프로그램의 실행을 중지한다. 단축키 : b
b r e a k *function-name*	프로그램이 *function-name*의 함수를 실행하기 시작하면 프로그램의 실행을 중지한다.
break *line-of-function* if *condition*	프로그램이 실행되는 코드가 *line-of-function*일 때 해당 조건이 만족되면 프로그램의 실행을 중지한다. 예: (gdb) break 46 if testsize == 100
break *routine-name*	프로그램이 어느 특정 루틴으로 진입시에 프로그램의 실행을 중지한다.
break *filename:line* break *filename:func*	프로그램이 여러 개의 파일로 이루어져 있을 때, 파일명이 *filename* 인 소스 파일의 *line* 의 코드를 수행할 경우 프로그램의 실행을 중지한다. *func* 도 마찬가지로 해당 함수로 진입할 경우 프로그램의 실행을 중지한다.
watch *expr*	어느 특정 조건이 만족되면 프로그램의 수행을 중지한다. 예: (gdb) watch testsize > 100000
continue	프로그램의 실행을 재개한다.
info breakpoints	현재 설정되어 있는 breakpoint들에 대한 정보를 출력한다.
disable *number*	해당 *number* 의 breakpoint를 억제시킨다.
enable *number*	해당 *number* 의 breakpoint를 작동시킨다.
enable once *number*	해당 *number* 의 breakpoint를 일시적으로 작동시킨다. 한번 작동 이후에는 자동으로 해당 breakpoint가 억제된다.

▪ 변수 관찰 및 값 설정

표 6. Inspecting and assigning values to variables.

명 령	설 명
whatis *var*	*var* 의 타입(type)을 출력한다.
ptype *struct*	구조체 *struct* 멤버들의 definition 을 출력한다.
set variable *assign*	변수에 값을 할당한다. 예: (gdb) set variable testsize = 20
display *var*	프로그램이 실행을 중지할 때마다 변수 *var* 의 값을 출력한다.

▪ 한 단계 실행

표 7. Single-step execution.

명 령	설 명
step	한 라인 실행 후 중지한다. 단 해당 라인에 사용자가 정의한 함수가 있으면, 해당 함수로 진입한다.
next	한 라인 실행 후 중지한다.

▪ 함수 호출

표 8. Single-step execution.

명 령	설 명
call *name*	*name*이란 이름의 함수를 호출하여 실행한다.
finish	현재 실행하고 있는 함수를 끝내고, return value가 존재하면 출력한다.
return *value*	현재 실행하고 있는 함수의 실행을 중지하고, return value로 *value*를 리턴 한다.

▪ call stack 이동

표 9. Moving up and down the call stack.

명 령	설 명
where	현재의 call stack에 대한 정보를 출력한다.
up [*n*]	현재의 call stack 위치에서 *n* 만큼 위로 stack frame을 이동한다. up이나 down 명령은 주로 해당 scope의 변수를 살펴볼 때 사용한다.
down [*n*]	현재의 call stack 위치에서 *n* 만큼 아래로 stack frame을 이동한다.
frame	현재의 stack frame에 대한 정보를 출력한다.

2-2-4. gdb를 이용한 Debugging 예

여기에서는 하나의 예제를 가지고 gdb를 사용하는 방법을 보이도록 하겠다. 우리가 테스트 해 볼 프로그램은 다음과 같다.

¤ test.c

```
#include   <stdio.h>
int mult(int a, int b){
    printf("%d x %d = %d\n", a, b, a*b);
}
int main(void){
    int i, j;
    printf("Test Program\n");
    for(i = 2; i < 10; i++){
        for(j = 2; j < 10; j++){
            mult(i, j);
        }
        printf("\n");
    }
}
```

이 파일을 다음과 같이 컴파일하고 gdb를 실행한다.

☞ 예제

```
$ gcc -o test -g test.c
$ gdb test
GNU gdb 5.0rh-5 Red Hat Linux 7.1
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

컴파일 시에 -g 옵션을 주는 것을 주의하자. 자 이제 디버깅을 시작하기 위하여 프로그램의 내부를 들여다보자.

☞ 예제

```
(gdb) l
1       #include        <stdio.h>
2
3       int mult(int a, int b){
4               printf("%d x %d = %d\n", a, b, a*b);
5       }
6
7       int main(void){
8               int i, j;
9
10              printf("Test Program\n");
(gdb)
```

이제 breakpoint를 걸어서 프로그램을 실행 도중에 중단시켜 보도록 하자.

☞ 예제

```
(gdb) l main
2
3       int mult(int a, int b){
4               printf("%d x %d = %d\n", a, b, a*b);
5       }
6
7       int main(void){
8               int i, j;
9
10              printf("Test Program\n");
11              for(i = 2; i < 10; i++){
(gdb) b 10
Breakpoint 1 at 0x804848a: file test.c, line 10.
(gdb) run
Starting program: /home/grad/lazylune/a/test

Breakpoint 1, main () at test.c:10
10              printf("Test Program\n");
(gdb)
```

먼저 breakpoint를 설정할 부분을 살펴보기 위하여 l main 명령을 사용하였다. 이 명령

은 main 함수를 화면에 출력해 준다. 여기에서는 breakpoint를 for문이 시작하기 전 라인(라인 넘버 : 10)에 설정하였다. breakpoint를 설정하고 **run** 명령으로 실행하면 breakpoint에서 프로그램이 중단되고 gdb 프롬프트가 나타난다. 이제 for문을 한라인 씩 수행하면서 i와 j값의 변화를 살펴보기로 하자.

☞ **예제**

```
(gdb) display i
1: i = 134518368
(gdb) display j
2: j = 134518152
(gdb) next
Test Program
11            for(i = 2; i < 10; i++){
2: j = 134518152
1: i = 134518368
(gdb)
12                  for(j = 2; j < 10; j++){
2: j = 134518152
1: i = 2
(gdb)
13                          mult(i, j);
2: j = 2
1: i = 2
(gdb)
2 x 2 = 4
12                  for(j = 2; j < 10; j++){
2: j = 2
1: i = 2
(gdb)
```

display 명령을 사용하면 해당 변수의 값을 프로그램 수행 중단 시 마다 살펴 볼 수 있다. 10 라인에서 처음으로 display를 사용하여 i, j 값을 살펴보면, 아직 초기화가 되지 않았기 때문에 알 수 없는 값이 들어가 있는 것을 확인할 수 있다. **next** 명령을 실행 하면 다음 라인의 코드가 수행 되면서 "Test Program"이란 메시지가 출력되는 것을 확인할 수 있다. **next** 명령 대신에 **step** 이란 명령을 사용할 수 도 있다. 이 두 명령은 mult(a, b)와 같이 프로그램 내에 정의된 함수를 처리하는데 차이가 있다. next는 내장 함수를 실행하여 결과를 출력하는 반면에 step은 mult(a,b) 함수 내부로 들어가서 함수 내부의 실행 과정을 살펴 볼 수 있다. next 명령 실행 이후에는 엔터를 쳐주면 자동으로 전에 실행했던 명령이 재실행 된다.

이제 i와 j값을 변경하여 mult() 함수를 실행해 보자.

☞ **예제**

```
(gdb) set variable i = 4
(gdb) set variable j = 5
```

```
(gdb) next
4 x 5 = 20
12                  for(j = 2; j < 10; j++){
2: j = 5
1: i = 4
(gdb) n
13                          mult(i, j);
2: j = 6
1: i = 4
(gdb) call mult(i, j)
4 x 6 = 24
$2 = 11
(gdb) call mult(j, i)
6 x 4 = 24
$3 = 11
(gdb)
```

set variable 명령은 변수의 값을 변경시켜 준다. 값이 변경된 이후 변경된 값을 가지고 call 명령으로 mult() 함수를 호출하면 프로그램의 실행 흐름과 상관없이 mult() 함수를 검증할 수 있다.

이러한 방식으로 프로그램의 흐름이나 변수의 값을 확인하면서 프로그램이 제대로 동작하는지를 살펴볼 수 있다.

2-3. make

2-3-1. 개요

make를 사용하는 주된 이유는 다수의 소스 파일들을 컴파일하기 위해서이다. 많은 사람들은 작은 프로그램을 작성할 때 단순히 수정하고 나서 모든 파일을 다시 컴파일하여 어플리케이션을 재구성한다. 그러나 좀더 규모가 큰 프로그램에서는 이런 간단한 방법이 약간의 문제를 가져올 것이다. 수정하고 컴파일하고 테스트하는 과정은 상당히 많은 시간을 소모할 것이다. 아주 작은 파일 하나를 수정하더라도 이를 위해 모든 파일들을 다시 컴파일해야 하는 상황이 발생할 것이다. make 프로그램은 프로그래머를 이러한 상황에서 해방시켜 준다.

make 명령은 많은 기능을 제공하지만, 어플리케이션의 구성 방법을 자동으로 인식하거나 찾아내지는 않는다. 따라서 어플리케이션의 구성 방법을 make에게 알려주는 파일을 제공해야 하는데 이 파일을 메이크파일(makefile)이라고 한다.

2-3-2. 실행 방법

$ 실행 방법

```
make [ -f makefile ] [ option ] ...  target ...
```

2-3-3. makefile의 형식

¶ 문법

```
target name: list-of-dependencies
    rules
```

makefile은 의존성과 규칙으로 구성된다. **의존성(dependency)**은 생성되는 파일인 대상과 이것이 의존하는 소스 파일을 포함한다. **규칙(rules)**은 의존하는 파일로부터 대상 파일을 생성하는 방법을 설명한다. **대상(target)**은 일반적으로 실행 파일이다.

makefile은 make 명령에 의해 사용된다. make는 구성할 대상 파일을 결정하고 나서 대상을 구성하기 위해 사용할 필요가 있는 규칙을 정하기 위해 소스 파일의 날짜와 시간을 비교한다. 종종 최종 대상 파일을 만들기 전에 중간 파일을 생성해야 한다. make 명령은 대상을 만드는 순서와 사용할 규칙의 정확한 순서를 결정하기 위해 makefile을 사용한다.

▪ 의존성

의존성은 최종적인 어플리케이션에서 각 파일이 소스 파일과 관련되는 방법을 지정한다. 가령 헤더파일 a.h, b.h, c.h와 다음과 같은 C 소스 파일 main.c, 2.c, 3.c가 있다고 하자.

¤ main.c

```
#include "a.h"
...
```

¤ 2.c

```
#include "a.h"
#include "b.h"
...
```

¤ 3.c

```
#include "b.h"
#include "c.h"
...
```

이 예제에서 최종 어플리케이션은 main.o, 2.o, 3.o를 요구(의존)한다. 또한 main.c는 a.h를 요구하고, 2.c는 a.h와 b.h를 요구하며, 3.c는 b.h와 c.h를 요구한다. 즉 컴파일을 하기 위해서는 그와 같은 파일이 존재해야만 한다. main.o는 main.c와 a.h의 변경에 의해 영향을 받고, 이런 두 파일의 하나가 변경되면 main.c를 다시 컴파일하여 생성할 필요가 있다. 이러한 의존성을 makefile에서는 다음과 같이 표시해 준다.

¤ 의존성 목록

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

이는 myapp가 main.o, 2.o, 3.o에, main.o가 main.c, a.h에 의존한다는 것을 알려준다.

이런 의존성은 소스 파일의 관련성을 보여주는 계층을 구성한다. 그래서 b.h가 변경되면 2.o와 3.o를 모두 재구성할 필요가 있고, 2.o와 3.o가 변경되었으므로 또한 myapp를 재구성할 필요가 있음을 쉽게 알 수 있다.

- 규칙

규칙은 대상을 생성하는 방법을 설명한다. 앞의 예제에서 make 명령이 2.o를 재구성할 필요가 있다면 어떤 명령을 사용해야 할까? 단순히 gcc -c 2.c를 사용하는 것으로 충분할 것이다. 사실 make는 많은 기본적인 규칙을 알고 있다. 그러나 include 디렉토리를 지정할 필요가 있거나, 디버깅을 위해 -g 옵션을 설정할 필요가 있다면 어떨까? makefile에서 분명한 규칙을 지정하여 설정할 수 있다.

위의 예제에 규칙을 적용하여 makefile을 만들어 보면 다음과 같다.

¤ makefile

```
myapp: main.o 2.o 3.o
    gcc -o myapp main.o 2.o 3.o
main.o: main.c a.h
    gcc -c main.c
2.o: 2.c a.h b.h
    gcc -c 2.c
3.o: 3.c b.h c.h
    gcc -c 3.c
```

- 매크로

¶ 문법

정의

MACRONAME = macro-body

사용

$(*MACRONAME*)

${*MACRONAME*}

make는 매크로를 지원하여 makefile을 일반적인 형식으로 작성할 수 있도록 해준다.

매크로는 종종 makefile에서 컴파일러의 옵션을 위해 사용된다. 어플리케이션은 보통 개발 과정에서는 최적화를 사용하지 않고, 디버깅 정보를 포함하는 상태로 컴파일을 하게 된다. 반면 최종 릴리즈 버전에서는 일반적으로 가능한 한 빨리 실행되고, 디버깅 정보를 가지지 않은 작은 바이너리 파일을 생성하게 된다. 매크로는 이것을 쉽게 변경하게 해준다.

기본적으로 makefile은 컴파일러가 gcc라고 가정한다. 그러나 다른 유닉스 시스템에서는 cc나 c89를 사용할 수도 있다. 따라서 다른 유닉스 버전으로 makefile의 이식을 원하거나, 기존의 시스템에서 다른 컴파일러를 사용한다면, 이를 위하여 매크로를 고쳐주면 된다.

매크로를 적용하여 앞에서 보인 makefile을 확장하면 다음과 같다.

¤ 확장된 makefile

```
# 컴파일러 이름  // '#'은 주석을 의미한다.
CC = gcc        // shell에서 변수(매크로)에 특정 값을 저장하는 것과 동일하다.
                // 이의 사용은 아래에 보인바와 같이 $(CC)로 사용한다.
# 헤더 파일의 위치
INCLUDE = .

# 개발용 옵션
CFLAGS = -g -Wall -ansi

# 출시용 옵션
#CFLAGS = -O -Wall -ansi  // 이 줄은 '#'으로 주석 처리되어 있다.
                          // 출시용이면 이를 지우고 위 개발용 옵션을 주석 처리한다.
myapp: main.o 2.o 3.o
    $(CC) -o myapp main.o 2.o 3.o
main.o: main.c a.h
    $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c
2.o: 2.c a.h b.h
    $(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c
3.o: 3.c b.h c.h
    $(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```

이와 같은 makefile을 개발 디렉토리에 만들어 주고 난 이후부터는, 프로그램을 컴파일하기 위해서는 단지 쉘에서 "make *target-name*" 명령만 실행해 주면 된다.

- 내장 규칙

make는 makefile을 단순화 하는 다수의 내장 규칙을 가진다. 내장 규칙은 사용자가 지정해 주지 않은, make 자체에서 사용하는 규칙들이다. 다음의 예제를 살펴보자.

☞ 예제

```
$ cat foo.c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    exit(EXIT_SUCCESS);
}
$ make foo
cc     foo.c   -o foo
$
```

여기에서 볼 수 있듯이, make는 gcc 대신에 cc를 선택했지만 컴파일러 호출 방법을 어느 정도 알고 있다. 이런 내장 규칙을 추론 규칙(inference rules)이라고 한다. 기본적인 규칙은 매크로를 사용하므로 매크로의 새로운 값을 지정하여 기본 동작을 변경할 수 있다.

-p 옵션을 통해서 make가 내장 규칙을 출력하도록 요청할 수 있다. 이러한 내장 규칙을 고려하여 오브젝트를 만들기 위한 규칙을 제거하고 의존성을 지정하여 makefile을 단순화할 수 있으므로 makefile의 관련 부분은 다음과 같다.

¤ makefile의 간소화

```
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

- 접미어

¶ 문법

```
.old_suffix.new_suffix:
```

makefile은 특정 문자로 끝나는 파일이 있을 때 다른 문자로 끝나는 파일을 생성하기 위해 사용할 규칙을 알 수 있게 하는 접미어(suffix)를 제공한다. 가장 흔한 규칙은 .c 파일로 .o 파일을 생성하는 규칙이다. 다음 예는 접미어를 사용하여 makefile을 줄인 예이다.

¤ 확장된 makefile

```
# .c와 .o 확장자에 대한 접미어 규칙을 제시
.SUFFIX: .c .o

CC = gcc
CFLAGS = -g -Wall -ansi
OBJFILES = main.o 2.o 3.o

# 접미어 규칙
.c.o:
    gcc -c -o $@ $(CFLAGS) $<  // '$<' : 확장자가 .c인 파일
                               // '$@' : 확장자가 .o인 생성될 파일.
myapp: $(OBJFILES)
    gcc -o myapp $(OBJFILES)

main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

위에서 사용된 $<는 특수문자로 '과거의' 접미어를 가지는 시작파일 이름으로 확장된다(즉, 확장자가 .c인 파일). 이 이외에 makefile에서 사용되는 특수 문자들은 다음과 같다.

표 10. 특수 문자.

변 수	설 명
$?	현재 대상보다 최근에 변경된 필수 조건의 목록
$@	현재 대상의 이름
$<	현재 필수 조건의 이름
$*	확장자를 제외하고 현재 필수 조건의 이름

3. UNIX C/C++ Programming 실습

3-1. 목 적

본 실험에서는 UNIX 환경에서 프로그램 작성(vi), 컴파일(gcc, make), 디버깅(gdb)하는 방법을 실습한다. 그런데 본 실험에서 주어진 문제는 경우의 수가 많으므로 실험 전 문제 및 풀이방법을 충분히 숙지하고 프로그램을 미리 작성한 후 실험에 임하도록 한다.

3-2. 프로그래밍 문제

fmt

UNIX도구 중에 하나인 fmt 명령은 텍스트를 읽어 각 줄을 일정 규칙에 따라 합치거나, 나누어 새로운 파일을 생성하는 프로그램이다. 한 줄은 특별한 경우를 제외하고 LIMIT=72자를 넘을 수 없다. 이를 위해 사용되는 규칙은 다음과 같은데 UNIX 본래의 fmt와 다소 다를 수 있다. 예는 편의상 LIMIT를 72가 아닌 수로 설정하였을 때의 결과를 보일 수도 있다.

R1. 출력 각 줄의 마지막에는 blank(' ')를 만들지 않는다.

R2. LIMIT보다 작은 글자로 구성된 줄 L1에 이어 다음 줄 L2의 단어를 같은 줄로 출력할 때 L1의 끝에 하나 이상의 blank가 있어도 단 하나의 blank만을 출력한다.

☞ **예제 (LIMIT=72)**

입력

```
Programming needs logical thinking.     \n
I like programming.          \n
```

출력

```
Programming needs logical thinking. I like programming.\n
```

예에서 '\n'는 줄바꿈 문자이다.

R3. 단어 사이의 빈칸의 개수는 그들 사이에서 줄이 바뀌지 않는 한 그대로 출력한다.

R4. 만일 문자수가 LIMIT보다 커서 줄을 나누어야 할 경우 단어와 단어 사이를 나누는데 그 사이에 존재하는 blank 글자는 출력하지 않으며, 새 줄로 출력되는 단어의 앞에도 blank를 붙이지 않는다.

☞ **예제 (LIMIT=20)**

입력

```
Your program   must    be well organized.\n
You  need    a    good planning fot that.\n
```

출력

```
Your program   must\n
be well organized.\n
You  need    a    good\n
planning fot that.\n
```

R5. 입력줄의 첫 글자가 blank이면 앞줄과 합쳐지지 않게 한다. 만일, 줄의 첫 부분에 여러 개의 blank가 있으면 이 역시 줄을 바꾸어 새 줄에 출력하고 첫 부분의 blank는 첫 번째 blank를 포함해 그 개수만큼 그대로 출력한다.

R6. 단어 중간에서 줄바꾸지 않으며 만일 단어 하나가 LIMIT 보다 큰 글자들로 구성된 경우에는 이를 분리하지 않고 한 줄에 그대로 출력한다.

R7. 빈 줄은 그대로 빈 줄로 출력한다. 빈 줄에 있는 blank 문자들은 출력하지 않는다.

R8. 입력 마지막 줄의 끝에 '\n'이 없으면 출력도 마찬가지로 '\n'을 출력하지 않는다. 그러나 '\n'이 있을 경우에는 이를 출력한다.

R9. 크기가 0인 empty 파일인 경우에는 아무것도 출력하지 않는다.

☞ **예제 (LIMIT=36)**

입력

```
  Youneedtoconsidereverypossiblecases. Can you see?\n
Good data structure implies a good program.\n
   If your program does not work, use debugger.   \n
  \n
\n
I am sure that youlikeproblemsolvingbyprogramming.\n
Aggree?\0
```

출력

```
  Youneedtoconsidereverypossiblecases.\n
Can you see? Good data structure\n
implies a good program.\n
   If your program does not work,\n
use debugger.\n
\n
\n
I am sure that\n
youlikeproblemsolvingbyprogramming.\n
Aggree?\0
```

위 예에서 '\0'는 EOS(end of string)를 의미한다.

3-3. 실험 방법

3-3-1. 문제 해결

문제를 단순히 하기 위하여 입력 각 줄의 글자 수는 '\n'와 '\0'을 포함하여 256글자를 넘지 않는다고 가정한다. 이 가정이 없다면 프로그램은 상당히 복잡해진다. 다음과 같은 정의를 프로그램에서 사용하자.

```
#define BNUM 256  // 한 줄을 읽을 입력 버퍼의 크기.
#define LIMIT 72  // 출력 줄의 글자 수 제한(프로그램 작성 시 작게 설정하여 테스트)
```

많은 규칙이 주어졌는데 이를 모두 만족하는 프로그램을 한번에 작성하려하지 말고 top down 형태로 접근하여 프로그램 구성을 충분히 생각한 후에 각 규칙을 점차적으로 만족하도록 프로그램을 점진적으로 완성하여야 한다.

먼저, 현재 출력 중인 줄 L1외에 이어지는 다음 줄 L2의 상태를 알 필요가 있다. 예를 들어 R5의 경우 다음 줄의 첫 글자가 blank인지 아닌지에 따라 출력의 처리가 달라진다. 즉, 첫 글자가 blank이면 줄을 바꾸어야 한다. 또한, 다음 줄이 빈 줄인 경우에는 현재 출력 줄인 줄의 글자수가 LIMIT보다 작은 경우에도 줄을 바꾸어야 한다. 따라서 두 개의 입력 버퍼를 마련하여 다음의 형태로 main 프로그램을 작성하고 나머지 기능을 완성한다.

```
void main(int argc, char *argv[]) {
  char *L1, *L2, *tmpline;
  Allocate memory for L1[] and L2[];
  ...
  if ( fgets(L1, BNUM, fp) == NULL ) // 한 줄을 읽는다.
    return; // 만일 빈 파일이면 그대로 프로그램을 끝낸다.
  ...       // L1에 대하여 필요한 전처리 과정을 수행한다.
  while ( 1 ) { // EOF가 나올때 까지 이 루프의 수행을 계속한다.
    ...     // L1에 대한 처리를 수행한다.
    if ( fgets(L2, BNUM, fp) == NULL ) // 다음줄을 읽어 L2에 저장한다.
      break;  // EOF이면 루프의 수행을 종료한다.
    ...       // L2에 대한 전처리, 첫글자 조사 등과 같은 일을 수행한다.
    tmpline = L1;   // 이 세 줄은 읽은 L2의 글자를 L1으로 복사하는 기능이다.
    L1 = L2;        // 단순히 포인터만 바꾸어 실제 글자 복사를 피한다.
    L2 = templine;
  }
}
```

파일에서 한 줄 L을 읽으면 L의 끝에 blank가 연속으로 있을 수 있다. 이러한 blank들은 필요 없는 글자들이고 프로그램 작성을 복잡하게 하므로 미리 제거해 준다.

☞ **예제**

입력

```
Programming needs  logical thinking.     \n
```

전처리 후

```
Programming needs  logical thinking.\n
```

입력

```
     \n
```

전처리 후

```
\n
```

프로그램 control을 위하여 다음과 같은 변수와 flag을 정의할 필요가 있다.

Count : 현재 줄에 실제로 출력할 글자의 개를 저장하고 있으며 글자 하나를 출력할 때마다 하나씩 그 값이 증가하도록 한다.

B_Flag : 규칙 R2의 적용에 필요하다. 즉, 현재 줄을 완전히 출력하였는데 Count≠0이라면 계속 같은 줄에 출력할 수 있으므로 B_flag을 1로 set하여 이를 수행토록 한다. 만일 다음 줄이 빈 줄이거나 첫 글자가 blank이면 줄을 바꾸어야하므로 B_flag을 0으로 set하여 줄바꿈을 수행하도록 한다. 그림 2에 이러한 프로그램 구성을 보인다.

그림 2에서 'output L1'은 줄 L1을 출력하는 함수로 뒤에 이의 구성에 대하여 설명한다. 이 함수에서 Count 값이 조정되며 B_Flag 역시 값이 조정되거나 사용된다. L1을 처리한 후에 Count 값을 보고 B_Flag의 set 여부를 추가로 결정한다. 'read L1'과 'read L2'는 한 줄을 읽는 함수이다. 한 줄을 읽은 후에는 위에서 설명한 줄 끝의 blank를 없애는 과정을 함수로 구현하여 반드시 실행하여야 한다. 만일 다음 줄 L2의 첫 글자가 ' '이거나 또

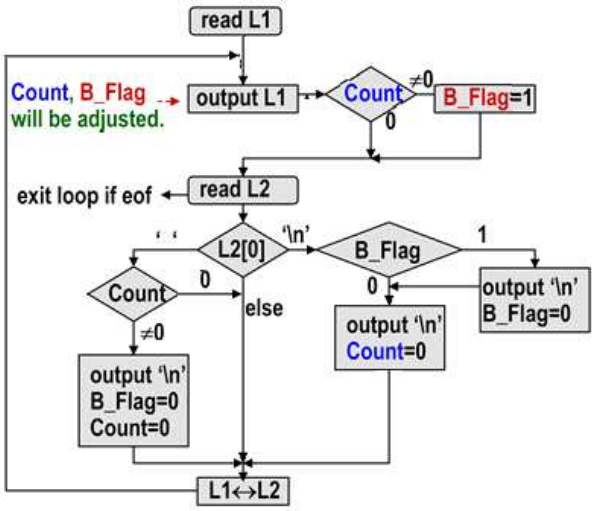

그림 2. 초기 프로그램 구성도.

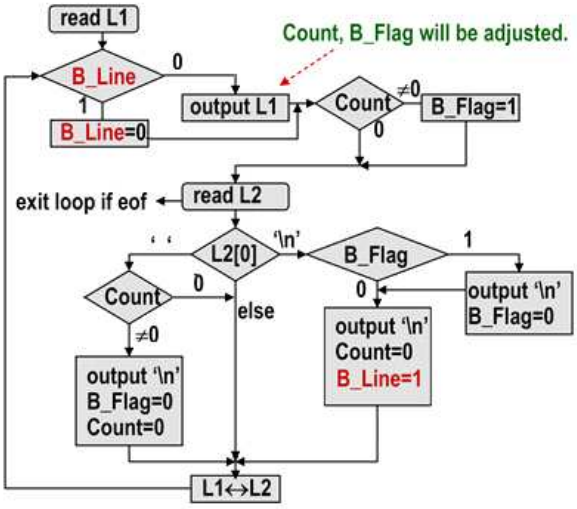

그림 3. B_Line flag를 반영한 프로그램 구성도.

는 '\n'인 경우에는 줄을 바꾸어 새 줄에 출력하여야 하므로 ' ' 를 삽입할 필요가 없어 ('output L1' 함수) '\n'를 하나 출력하고(즉, 줄 바꾸고) B_Flag, Count의 값을 0으로 한다. 그림 2의 기능을 앞에서 보인 main( ) 함수의 적절한 위치에 구현하도록 하자.

B_Line : 한 줄을 읽고 뒤의 blank 글자를 제거하는 전처리 과정을 완료한 후에 그 줄이 빈 줄이라면 B_Line flag을 set 한다. 프로그램 작성시 B_Line이 1이라면 'output L1' 을 호출할 필요 없이 다음 단계를 수행하도록 한다. 이를 반영하여 그림 2의 구성을 수정하여 그림 3에 보인다. 따라서 한 줄을 매번 읽을 때마다 읽은 줄이 빈줄인지 아닌지를 조사하여 B_Line flag을 조정한다.

텍스트 파일의 마지막 줄이 빈 줄인 경우 'output L1' 함수에서는 아무것도 출력되지 않으면 다음 줄 L2를 읽을 때 EOF가 검출되어 루프의 수행을 종료하게 된다. 따라서, 마지막 빈 줄을 출력하지 못하는 결과를 초래하는데 이를 반영하기 위하여 파일을 읽는 while loop가 종료된 후에 마지막 줄이 빈 줄인지를 검사하여 이를 출력하는 후처리 과정을 추가 한다. 이는 L1 버퍼의 첫 글자를 검색하여 쉽게 해결할 수 있다. 그림 4에 그림 3의 과

정에 이 후처리 과정을 추가한 프로그램의 구성을 보인다. 그리고 그림 5에 지금까지 기술한 내용을 종합한 전체 프로그램 구성을 보인다. 그림 5에서 'output L1' 함수만 구성하면 전체 프로그램 구성이 완료된다.

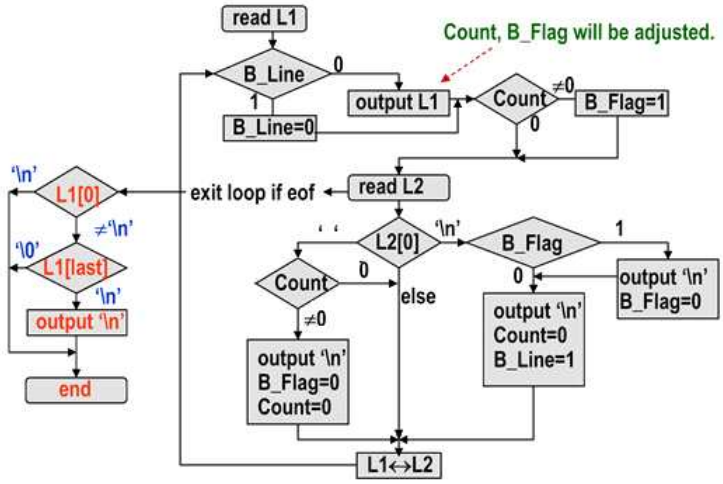

그림 4. 마지막 빈 줄의 처리 과정을 포함한 프로그램 구성.

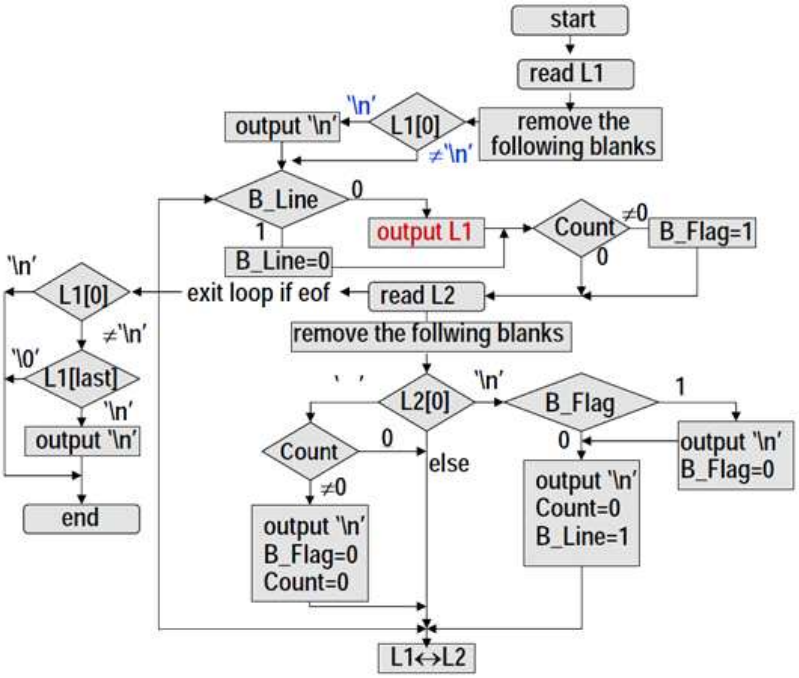

그림 5. 전체 프로그램 구성도.

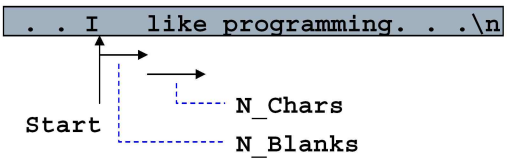

그림 6. N_Blanks, N_Chars.

마지막으로 ‘output L1' 함수의 구현에 대하여 알아보기로 하자. 단어 중간에서 줄을 바꿀 수 없기 때문에 단어별로 구분하여 이를 한번에 출력하여야 한다. 이를 위하여 그림 6에 보인 봐와 같이 Start, N_Blanks, N_Chars라는 세 개의 변수를 도입한다. Start는 현재 이 인덱스까지 글자의 출력이 진행되어 왔음을 의미한다. 이 값은 항상 단어의 끝을 가리키고 있다. N_Blanks는 Start 이후 존재하는 blank 글자의 개수를 나타내고 N_Chars는 다음 단어의 글자수를 나타낸다. 초기 ’output L1' 함수가 호출 될 때 Start 값은 0으로 초기화한다. 'output L1' 함수는 반복적으로 N_blanks와 N_Chars를 검출하여 이에 해당하는 글자를 N_Chars가 0이 될 때까지 반복한다. Start 값이 주어졌을 때 N_Blanks와 N_Chars 값을 구하는 함수 Get_Blanks_Chars()를 구현하여 이용하도록 하자. 그림 7에 ‘output L1' 함수의 전체 구성을 보인다.

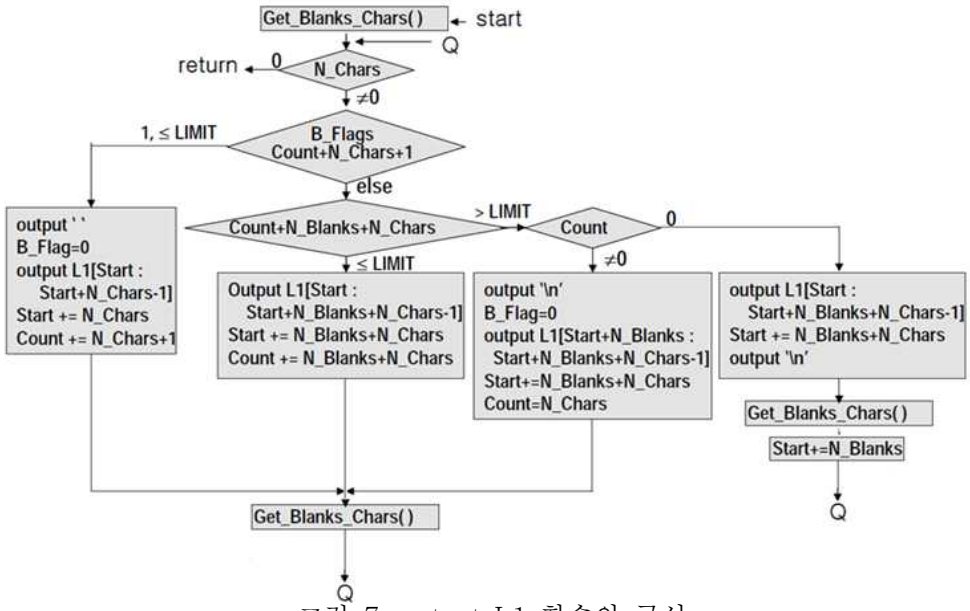

그림 7. output L1 함수의 구성.

3-3-2. 프로그래밍 편집 및 컴파일

각각의 프로그램 코드를 편집하고 이를 gcc를 사용하여 직접 컴파일 해본다.
여기에서 생성할 파일들은 다음과 같다.

표 11. 생성 파일.

파 일 명	설 명
main.c	main routine 이 들어가는 파일이다.
output.c	'output L1' 함수를 작성한다.
stringM.c	각종 스트링 관련 함수를 작성한다.
some header files	header 파일에 전역변수, define문 등을 정리한다.

이렇게 작성한 소스 코드를 gcc를 이용하여 다음과 같이 컴파일해 준다.

☞ 컴파일 예

```
$ gcc -c -g main.c
$ gcc -c -g output.c
$ gcc -c -g stringM.c
$ ls
main.c   main.o fmt.h   output.c   output.o
stringM.c   stringM.o
$ gcc -o fmt main.o output.o stringM.o
$ ls fmt
fmt
$
```

여러 소스 파일들을 하나의 프로그램으로 컴파일하기 위해서는 위의 예에서처럼 각각의 소스 파일을 object 파일로 만들어 준 뒤, 이들을 링킹하여 실행 파일을 생성한다.

3-3-3. makefile 작성

2-3 장을 참고하여 makefile을 작성한다. makefile 작성 이후에는 make 명령어를 실행해 컴파일이 되는지 확인 해 본다.

3-3-4. gdb를 이용한 디버깅

gdb를 이용하여 디버깅해 본다.

3-4. 숙제 및 결과 보고서 작성 내용

첨부한 예비보고서의 내용을 작성하여 실험 당일 제출한다. 숙제 및 결과 보고서 제출은 강의 일정에 따라 제출한다.

실험 UNIX-2 예비보고서

전공: 학년: 학번: 이름

1. 목 적

UNIX 상에서 제공하는 C/C++관련 도구를 미리 사용해 봄으로써, 수업시간에 실습이 원활히 진행될 수 있도록 한다.

2. 예비 학습

본문을 읽고 c/c++ 프로그램의 컴파일 과정에 대하여 요약하라. 각 단계별로 하는 일들과 관련된 도구들 또한 명시하라.

실험 UNIX-2 결과보고서

전공: 학년: 학번: 이름

1. 목 적

실습 과정에 개발한 fmt에 대하여 결과 보고한다.

2. 문제 풀이 결과

2-1. 알고리즘

fmt를 구현하기 위하여 사용한 알고리즘을 정리하여 기술하시오.

2-2. 테스트

조교가 제시한 테스트 데이터에 대한 출력 결과를 제출하시오. 제출방법은 조교의 지시에 따르시오.

참고 서적

1. Beginning Linux Programming, Richard Stones & Neil Matthew, WROX
2. Learning the bash shell, Cameron Newham & Bill Rosenblatt, O'Reilly
3. Programming with GNU Software, Mike Loukides and Andy Oram, O'Reilly
4. Debugging with GDB: The GNU Source-Level Debugger, Richard Stallman, FSF
5. RedHat Linux 9 Bible, Christopher Negus, John Wiley
6. UNIX in a nutshell A Desktop Quick Reference for SVR4 and Solaris 7 (3rd Edition), Arnold Robbins, O'Reilly
7. UNIX Power Tools Third Edition, Shelley Powers & Jerry Peek & Tim O'Reilly & Mike Loukides, O'Reilly