

Programming Assignment #0

컴퓨터구조 수업 수강을 환영합니다!

이번 과제는 컴퓨터구조 수업과 이후의 과제를 수행하기 전에 필요한 C programming Skill에 대해 평가 및 복습을 하는 것이 주 내용입니다. 과제는 A, B 두 개의 파트로 나누어져 있으며, 각 섹션에서는 각자 다른 부분을 평가합니다.

- Part A: 기본적인 포인터에 대한 이해와 응용 Skill
- Part B: File 입출력에 대한 이해와 문자열을 parsing할 수 있는 Skill

준비사항: 리눅스 환경 숙지! (아주BB 2주차 2회 및 3회 설명 동영상 듣고 시작하시길 바랍니다.)

Part A: 기본적인 포인터에 대한 이해와 응용 Skill

담당조교: 김종현

1. 개요

Part A에서는 포인터(pointer)에 대한 기본 개념과 이를 응용하여 구조체 포인터, 구조체 내부 변수에 포인터가 있는 경우 등을 알고있는지를 묻습니다.

구현해야 할 프로그램은 (x, y) 좌표 두개의 값을 인자로 받아서 x값들의 합, y값들의 차를 각각 출력(P1~P2)하고 두 점사이에 거리를 대각선으로 하는 직사각형의 넓이를 계산(P3~P6)하는 프로그램입니다. 추가적으로 문자열의 순서를 반대로 바꾸는 문제(P7)가 있습니다.

본 과제의 목표인 포인터의 개념을 잘 알고 있는지를 판단하기 위해서 첫번째 좌표는 int형 변수를 담고 있는 struct Point_val이라는 구조체를 사용하고, 두번째 좌표는 int*형 변수를 담고 있는 struct Point_ref 구조체를 사용해서 문제를 풀어야 합니다.

<pre>struct Point_val { int x; int y; }</pre>	<pre>struct Point_ref { int *x; int *y; }</pre>
---	---

프로그램을 실행할 때 두 점의 좌표 (x1, y1), (x2, y2)와 문자열이 쓰여져 있는 파일을 전달인자로 받아서 실행이 됩니다.

각 문제들은 main.c의 main 함수에 정의되어 있으며 문제를 해결하기 위한 함수들은 util.c에 작성되어 있습니다. util.c의 함수에 적절한 코드를 작성하여 문제들을 해결하는 프로그램을 작성하세요. (main.c의 main 함수를 수정하는 것이 아닙니다.)

2. 구현 사항

- P1: $x_1 + x_2$ 의 결과를 Call-by-Value로 구현하기
- P2: $y_1 - y_2$ 의 결과를 Call-by-Reference로 구현하기
- P3: struct Point_ref 구조체 변수를 struct Point_val 구조체 변수로 변환하기
- P4: struct Point_val 구조체 변수를 기반으로 한 직사각형의 면적을 계산하는 함수 구현
- P5: struct Point_val 구조체 변수를 struct Point_ref 구조체 변수로 변환하기
- P6: struct Point_ref 구조체 변수를 기반으로 한 직사각형의 면적을 계산하는 함수 구현
- P7: 문자열을 반대로 뒤집는 함수 구현(e.g. "abcd" → "dcba")
- 참고 사항: 두 점 사이에 생기는 사각형의 경우의 수는 가로와 세로가 좌표축에 평행한 경우로 제한한다.

예시) P1=(1, 3), P2=(4, 1)인 경우

가로: 두 점 간의 x좌표 값 차이 (e.g. $x_1=1, x_2=4$ 이므로, 가로 길이 = 3)

세로: 두 점 간의 y좌표 값 차이 (e.g. $y_1=3, y_2=1$ 이므로, 세로 길이 = 2)

면적 계산: $3 * 2 = 6$

3. 예시

3.1 제공되는 코드 다운로드

```
$] git clone https://github.com/csl-ajou/sce212-project0.git
Cloning into 'sce212-project0'...
remote: Enumerating objects: 48, done.
remote: Counting objects: 100% (48/48), done.
remote: Compressing objects: 100% (42/42), done.
remote: Total 48 (delta 1), reused 48 (delta 1), pack-reused 0
Unpacking objects: 100% (48/48), done.
Checking connectivity... done.
```

git을 통한 코드 다운로드 방법

3.2 소스 코드 컴파일

```
$] pwd
/home/csl
$] cd sce212-project0/PA0-A
$] pwd
/home/csl/sce212-project0/PA0-A
```

코드가 있는 위치로 이동

```
$] make
gcc -std=c99 -g -I ./ -Werror -c -o main.o main.c
gcc -std=c99 -g -I ./ -Werror -c -o util.o util.c
gcc -o pa0_a main.o util.o
```

make 명령어를 이용하여 소스 코드를 컴파일 하여 pa0_a 실행 파일을 생성한다.

3.3 결과 비교

```
$] make test_0
Testing example1
--- sample_output/example1.out 2021-03-10 14:58:01.478646992
+0000
+++ sample_input/example1.out 2021-03-10 15:02:35.209438611
+0000
@@ -1,6 +1,6 @@
  x1: 1, y1: 2, x2: 4, y2: 8, word:erutcetihcrA retupmoC ekil I
- sum_x: 5
- sub_y: -6
- calc_area1: 18
- calc_area2: 18
- The reverse is I like Computer Architecture
+ sum_x: 0
+ sub_y: 0
+ calc_area1: 0
+ calc_area2: 0
+ The reverse is erutcetihcrA retupmoC ekil I
Results not identical, check the diff output
```

Makefile에 정의해놓은 테스트 케이스 (example1 ~ example5)를 이용하여 reference 프로그램의 sample_output(---)과 비교하여 자신의 출력결과(+++)가 어디서 잘못되었는지 알 수 있다. 'make test' 명령어를 이용하면 테스트케이스 1번 부터 5번까지 모두 테스트한 결과를 보여준다.

만약 자신의 출력결과가 reference 프로그램의 출력과 동일하면 아래와 같이 나타난다. 아래는 test_1 에 대해서 수행한 결과 이고 이런식으로 5개의 test를 개별적으로 실행해볼 수도 있다.

```
$] make test_1
Testing example1
Test seems correct
```

3.4 참고: make test를 이용하지 않고 직접 실행파일을 이용하여 결과 확인

```
# Check your current directory path
$] pwd
/home/sce212/st201924165/PA0-A

# Execute your binary file with input arguments.
$] ./pa0_a sample_input/example1.in

# Output (This is not answer)
```

```
x1: 1, y1: 2, x2: 4, y2: 8, word:erutcetihcrA retupmoC ekil I
sum_x: 0
sub_y: 0
calc_area1: 0
calc_area2: 0
The reverse is erutcetihcrA retupmoC ekil I
```

4. 결과 제출

본 숙제에서 작성한 `util.c` 파일만 <https://sslabs.ajou.ac.kr/pasubmit>에서 제출한다. 제출하고 나서 반드시 Test를 통해서 본인이 테스트한 결과와 동일한 결과가 나오는지 확인을 진행한다.

Part B: File 입출력에 대한 이해와 문자열을 parsing할 수 있는 Skill

담당조교: 정진우

1. 개요

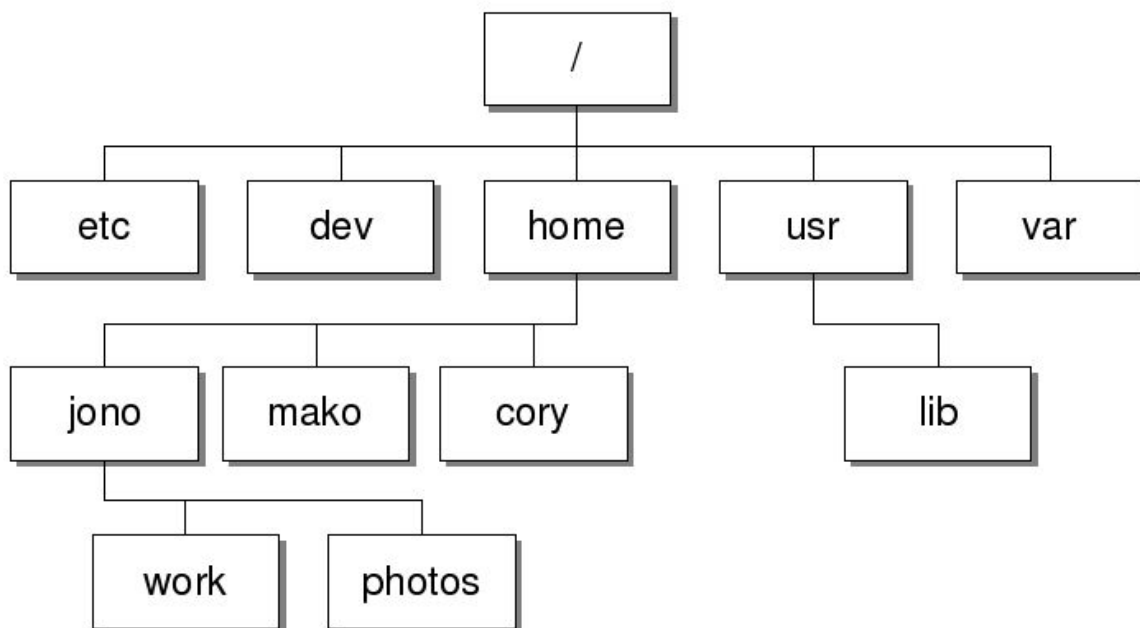
해당 프로젝트는 특정 디렉토리의 파일들을 출력하는 프로그램을 구현하는 것입니다. 이 프로젝트의 주 목적은 학생들에게 C programming language 에서의 문자열 파싱(parsing), 포인터(pointer), 구조체(structure)에 대한 이해를 돕기 위해서 준비하였습니다.

간략하게 구현할 프로그램을 설명하자면, 입력으로 파일들의 경로(path)가 주어지고, 이 정보를 가지고 특정 디렉토리 안에 있는 파일들을 출력하는 프로그램입니다.

2. 배경지식

2.1 디렉토리(directory) 구조

Linux는 기본적으로 사용자가 볼 수 있는 모든 것부터 보이지 않는 운영체제 및 하드웨어가 구성하고 있는 모든 것까지 모두 파일(file) 형태로 이루어져있습니다. (Linux 자체가 파일의 총 집합체라고 말해도 과언이 아닙니다.) 아래는 Linux kernel 이 구성하고 있는 디렉토리 구조입니다.



몇 가지 생략된 디렉토리 가 있지만 일반적으로 그림과 같은 계층적 구조를 이루고 있습니다. 모든 파일이나 디렉토리들은 root라고 불리는 디렉토리에서 부터 시작하며, '/' 로 표현됩니다.

2.2 경로 (Path)

Linux에는 Windows 와 동일하진 않지만 거의 비슷한 개념으로 경로(Path) 라는 것이 있습니다. 예를 들어 Windows 에서 "C:\Program files\AppData\" 로 path 를 나타낼 수 있다면 Linux 에선 비슷하게 "/home/user/application" 의 형태로 나타낼 수 있습니다. 경로를 나타내는 방식은 두 가지가 있는데 각각 표기하는 방법에 따라 불려지는 이름이 다릅니다.

- **Absolute path**: 직역하면 절대 경로라는 뜻으로 root부터 선택된 파일, 디렉토리 까지의 전체 경로를 뜻합니다. 예를 들어 앞의 예제와 같은 것이 absolute Path 입니다.
- **Relative path**: 직역하면 상대 경로라는 뜻으로 선택된 파일 및 디렉토리의 시점에서 보여지는 경로를 의미합니다.

이번 과제에서는 Path를 상대 경로 없이 **절대 경로**만을 사용하여 표현하였습니다.

3. 구현 사항

- P1: `utils.c`의 `parse_str_to_list()` 작성.
이 함수는 입력 버퍼를 '/' 문자 또는 개행문자를 단위로 하여 tokenize시키고, 그 결과로 만들어진 token을 `token_list`에 넣는 함수이다.

hint: `strtok()` 함수를 사용하면 쉽게 문자열을 tokenize할 수 있으며, `token_list`의 마지막 인덱스는 파일 이름, 나머지 인덱스는 디렉토리 이름이 담긴다.

예시)

파일 입력: "/home/user/text.txt"

token_list:

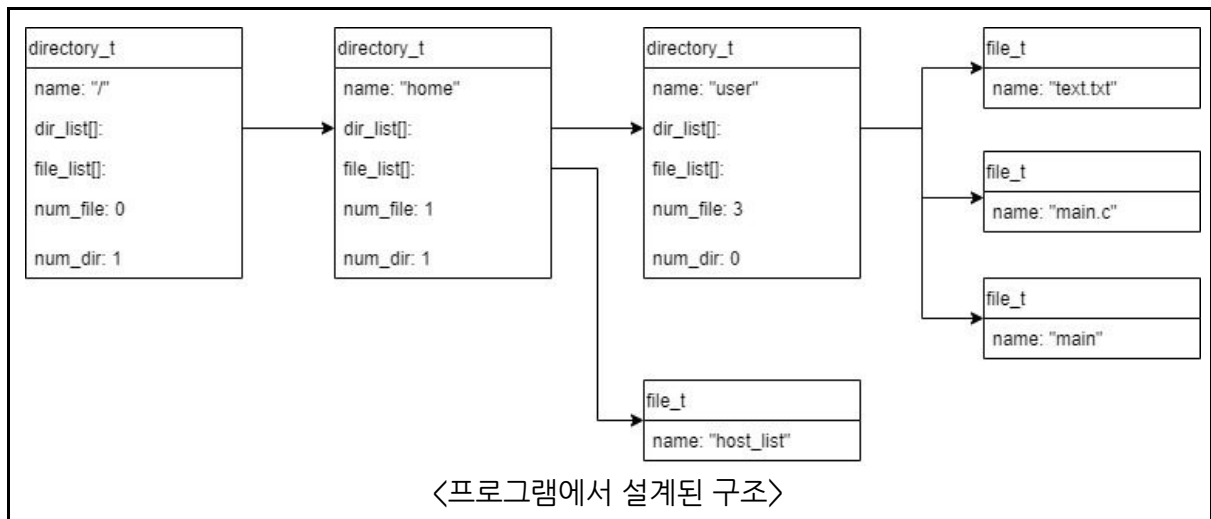
"home"	"user"	"text.txt"
--------	--------	------------

- P2: `dir_file.c`의 `make_dir_and_file()` 작성.
이 함수는 `root_dir` 디렉토리를 시작으로하여 `token_list`에 있는 token을 이용하여 디렉토리와 파일들의 계층적 구조로 만드는 함수이다.

예시)

```
/home/user/text.txt
/home/user/main.c
/home/user/main
/home/host_list
```

<파일들의 경로>



3.1 입력

dir_input	file_input
/home/user /home/	/home/user/text.txt /home/user/main.c /home/user/main /home/host_list

입력은 두 개의 텍스트 파일로 받으며, dir_input과 file_input이 있습니다.

- dir_input: 디렉토리들의 절대 경로를 담은 텍스트 파일
- file_input: 파일들의 절대 경로를 담은 텍스트 파일

위 예제의 경우, '/home/user/' 디렉토리안에 text.txt, main.c, main 파일이 존재하고, '/home/' 디렉토리안에는 host_list 파일이 존재하는 경우입니다.

3.2 출력

3.1에서 사용한 예시를 입력으로 하였을 때, 나오는 출력은 아래와 같습니다.

```
text.txt
main.c
main
host_list
```

여기서 주의할 점은 **디렉토리가 아닌 파일만을 출력한다는 점입니다**. 또한, 디렉토리는 존재하지만 그 안에 파일이 존재하지 않을 경우에는 출력하지 않습니다.

출력 순서는 다음과 같습니다.

1. dir_input에 적힌 디렉토리 순서로 출력
2. 디렉토리 내부에 여러 파일이 존재하는 경우, file_input에 적힌 순서로 출력 (알파벳 순서가 아님)

4. 예시

4.1 제공되는 코드 다운로드 (PA0-A에서 했다면 무시하고 넘어감)

```
$] git clone https://github.com/csl-ajou/sce212-project0.git
Cloning into 'sce212-project0'...
remote: Enumerating objects: 48, done.
remote: Counting objects: 100% (48/48), done.
remote: Compressing objects: 100% (42/42), done.
remote: Total 48 (delta 1), reused 48 (delta 1), pack-reused 0
Unpacking objects: 100% (48/48), done.
Checking connectivity... done.
```

git을 통한 코드 다운로드 방법

4.2 소스 코드 컴파일

```
$] pwd
/home/csl

$] cd sce212-project0/PA0-B
$] pwd
/home/csl/sce212-project0/PA0-B
```

코드가 있는 위치로 이동

```
$] make
gcc -std=c99 -g -I ./ -I ./include -c -o utils.o utils.c
gcc -std=c99 -g -I ./ -I ./include -c -o main.o main.c
gcc -std=c99 -g -I ./ -I ./include -c -o dir_file.o dir_file.c
gcc -g -o pa0_b ./utils.o ./main.o ./dir_file.o
```

코드를 make를 이용하여 컴파일 하여 pa0_b 실행파일을 생성한다.

4.3 결과 비교

```
$] make test_0
Testing example0
--- my_outputs/output0      2020-03-20 14:37:18.789646635 +0900
+++ sample_outputs/output0 2020-03-20 14:37:09.329573792 +0900
@@ -0,0 +1,5 @@
+main.c
+README.md
+.gitignore
+figure1.png
+figure2.png
      Results not identical, check the diff output
```


Makefile에 정의해놓은 테스트 케이스 (test_0 ~ test_4)를 이용하여 reference 프로그램의 output과 비교하여 자신의 출력결과가 어디서 잘못되었는지 알 수 있다. 'make test' 명령어를 이용하면 테스트케이스 0번 부터 4번까지 모두 테스트한 결과를 보여준다.

만약 자신의 출력결과가 reference 프로그램의 출력과 동일하면 아래와 같이 나타난다.

```
$] make test_0
Testing example0
    Test seems correct
```

4.4 Make를 이용하지 않고 직접 실행파일을 이용하여 테스트

```
$] ./pa0_b sample_inputs/input0/file_input \
> sample_inputs/input0/dir_input

main.c
README.md
.gitignore
figure1.png
figure2.png
```

5. 결과 제출

본 숙제에서 작성한 utils.c 그리고 dir_file.c 파일 2개를 zip 혹은 tar.gz 형태로 압축하여 제출한다.

```
$] pwd
/home/cs1/sce212-project0/PA0-B

$] make submission
Generating a compressed file (pa0-b-submission.tar.gz) including
utils.c and dir_file.c
a utils.c
a dir_file.c

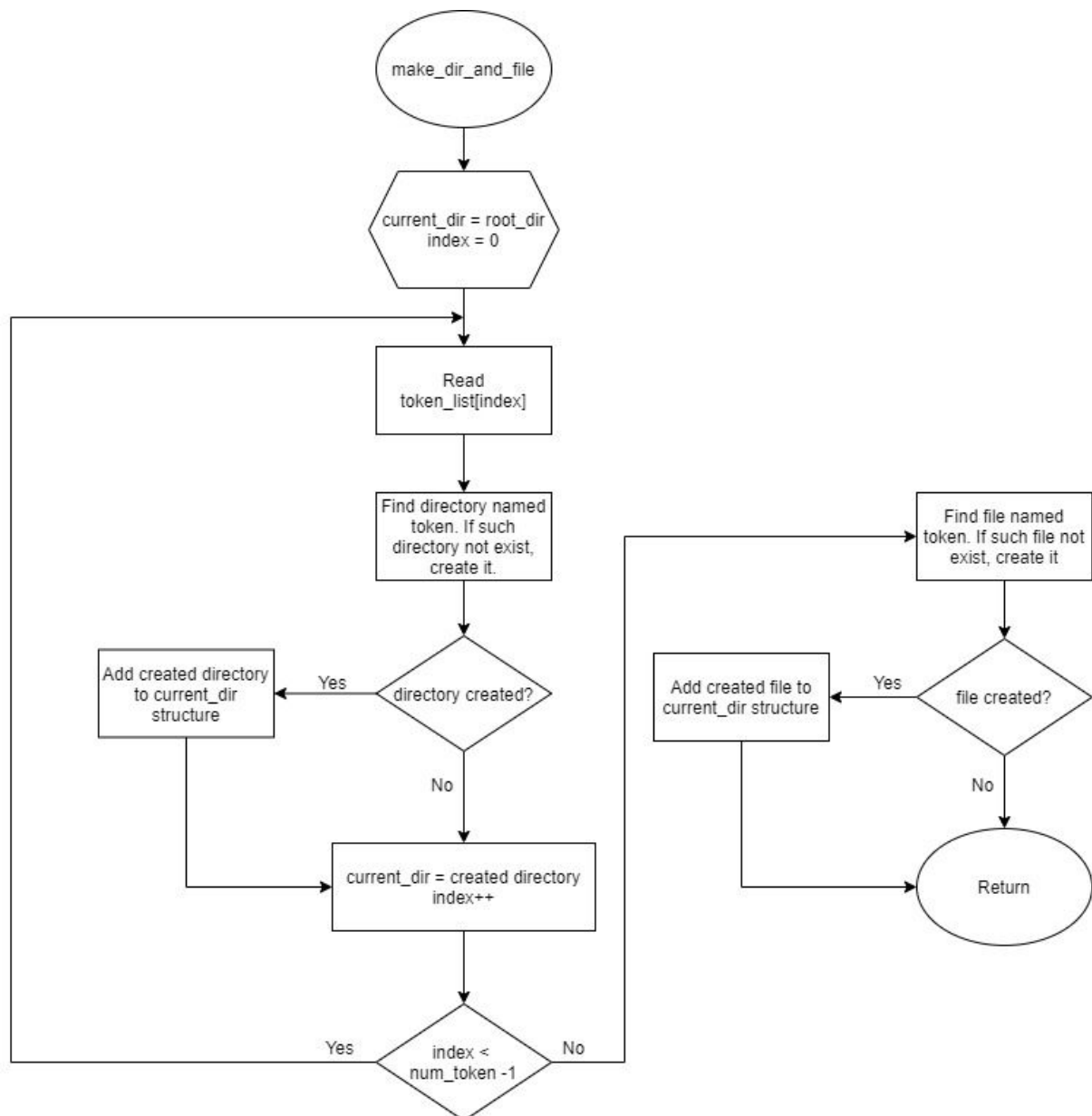
$] ls pa0-b-submission.tar.gz
pa0-b-submission.tar.gz
```

만들어진 pa0-b-submission.tar.gz 파일을 <https://sslab.ajou.ac.kr/pasubmit>에서 제출하고 나서 반드시 Test를 통해서 본인이 테스트한 결과와 동일한 결과가 나오는지 확인을 진행한다.

6. 참고

아래는 구현을 할 때 참고하면 되는 부분이고 본인의 방법으로 진행해도 상관 없음

6.1 make_dir_and_file에 대한 Flow Chart



6.2 Segmentation Fault가 발생했을 때, 디버깅 방법

코드를 작성하다 보면 Segmentation Fault 라는 에러를 만나실 수 있는데, 해당 에러가 발생했을 때 디버깅하는 법을 알려드리려고 합니다. 우선 Segmentation Fault에 대해서 소개한 뒤 디버깅 방법에 대해서 설명하겠습니다.

6.2.1 Segmentation Fault 이란?

Segmentation Fault는 컴퓨터 소프트웨어의 실행 중에 일어날 수 있는 특수한 오류이며, 발생 원인은 프로그램이 허용되지 않은 메모리 영역에 접근을 시도하거나, 허용되지 않은 방법으로 메모리 영역에 접근을 시도할 경우 발생합니다.

예를 들어, 포인터 변수로 부터 할당하지 않은 메모리를 접근했을 때 또는 할당 범위를 넘어서 접근했을 때 주로 발생합니다.

6.2.2 Segmentation Fault 발생 원인 찾기

Ubuntu Linux를 사용할 경우 아래의 명령어를 통해서 valgrind¹ 를 설치해야합니다.

```
$] sudo apt-get update && sudo apt-get install valgrind
```

설치가 완료됐으면 아래 명령어를 통해서 오류 원인과 오류가 발생한 코드 라인을 확인할 수 있습니다.

```
$] pwd
/home/csl/sce212-project0/PA0-B
$] make memory_check test=<example number>

# 만약 example 0번에 대해서 디버깅하고 싶다면, make memory_check
test=0 으로 입력하고, example 1번에 대해서 디버깅하고 싶다면, make
memory check test=1 를 입력하면 됩니다.
```

실행 예시)

```
$] make memory_check test=1
valgrind ./pa0_b sample_inputs/input1/file_input
sample_inputs/input1/dir_input
==12621== Memcheck, a memory error detector
==12621== Copyright (C) 2002-2017, and GNU GPL'd, by Julian
Seward et al.
==12621== Using Valgrind-3.13.0 and LibVEX; rerun with -h for
copyright info
==12621== Command: ./pa0_b sample_inputs/input1/file_input
sample_inputs/input1/dir_input
==12621==
==12621== Use of uninitialised value of size 8
==12621==    at 0x4C32E00: strcpy (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==12621==    by 0x108D16: parse_str_to_list (utils.c:34)
==12621==    by 0x108F76: main (main.c:40)
==12621==
```

¹ Valgrind는 C/C++ 프로그램에서 발생 할 수 있는 메모리 누수 또는 오류 등의 문제를 찾을 수 있는 tool

```

==12621== Invalid write of size 1
==12621==    at 0x4C32E00: strcpy (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==12621==    by 0x108D16: parse_str_to_list (utils.c:34)
==12621==    by 0x108F76: main (main.c:40)
==12621== Address 0x0 is not stack'd, malloc'd or (recently)
free'd
==12621==
==12621== Process terminating with default action of signal 11
(SIGSEGV)
==12621== Access not within mapped region at address 0x0
==12621==    at 0x4C32E00: strcpy (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==12621==    by 0x108D16: parse_str_to_list (utils.c:34)
==12621==    by 0x108F76: main (main.c:40)
==12621== If you believe this happened as a result of a stack
==12621== overflow in your program's main thread (unlikely but
==12621== possible), you can try to increase the size of the
==12621== main thread stack using the --main-stacksize= flag.
==12621== The main thread stack size used in this run was
8388608.
==12621==
==12621== HEAP SUMMARY:
==12621==    in use at exit: 10,072 bytes in 5 blocks
==12621== total heap usage: 6 allocs, 1 frees, 14,168 bytes
allocated
==12621==
==12621== LEAK SUMMARY:
==12621==    definitely lost: 0 bytes in 0 blocks
==12621==    indirectly lost: 0 bytes in 0 blocks
==12621==    possibly lost: 0 bytes in 0 blocks
==12621==    still reachable: 10,072 bytes in 5 blocks
==12621==    suppressed: 0 bytes in 0 blocks
==12621== Rerun with --leak-check=full to see details of leaked
memory
==12621==
==12621== For counts of detected and suppressed errors, rerun
with: -v
==12621== Use --track-origins=yes to see where uninitialised
values come from
==12621== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0
from 0)
Makefile:63: recipe for target 'memory_check' failed
make: *** [memory_check] Segmentation fault

```