PA #0: Warming up C Programming

Understanding File I/O and String Parsing

Due: March 21, 11:59 PM

1. Overview

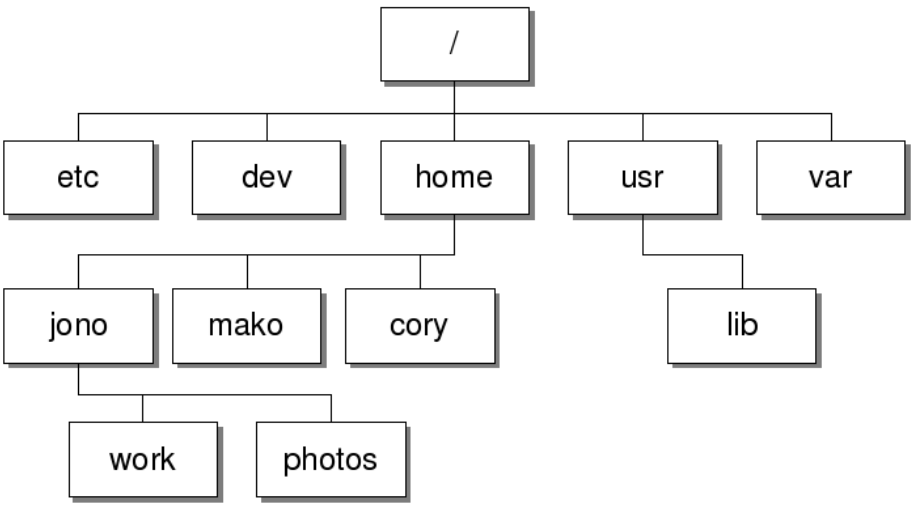
The goal of this assignment is to familiarize yourself with file I/O and string (characters) representations in C programming. If you are not accustomed to manipulating pointers in C, this assignment will help you understand how data, especially characters, are represented in memory. Additionally, you will gain experience working in a Linux environment using the shell and navigating the directory structure through the command-line interface (CLI).

To understand the directory structure and how characters (data type) are stored in memory, the assignment is to print out the files for a given directory. Note that we provide a list of directories and files which are not actual, but virtual. The skeleton code takes two input files: `dir_input` and `file_input`. `dir_input` contains the list of directories and `file_input` presents files in those directories. Section 2 provides the background of the directory structure in Linux and Section 4.3 provides more information about the skeleton code, and how it works.

2. Background

2.1 Directory structure in Linux

In Linux, the directory structure is hierarchical and starts from the **root directory (`/`)**, which contains all other directories and files. As shown in the following figure, under the root directory, it shows `etc`, `dev`, `home`, `usr`, and `var` directories.



The `home` directory contains user home directories. In the figure, there are three users: `jono`, `mako`, and `cory`. This structure helps organize the system efficiently, separating user files, system configurations, and software libraries. If you're new to Linux, exploring these directories using commands like `ls` and `cd` in the terminal can help you get familiar with the environment.

2.2 Path

In Linux (and other operating systems), the **path** is the address of a file or directory in the filesystem. For example, Windows represents paths as `C:\Program Files\AppData\`, whereas Linux uses a similar structure like `/home/user/application`.

There are two ways to represent a path:

- **Absolute path:** The complete path from the root directory to the file or directory (e.g., `/home/user/text.txt`).
- **Relative path:** The path relative to the current working directory.

This assignment will use absolute paths only.

3. Tasks

The assignment consists of two main tasks:

3.1 Task #1: Implement `parse_str_to_list()` in `utils.c`

```
// This function splits the input string (const char* str) to tokens
// and put the tokens in token_list. The return value must be the number
// of tokens in the given input string.
int parse_str_to_list(const char* str, char** token_list) {
     /* Fill this function */
}
```

This function tokenizes the first parameter (`str`) using `/` or a newline (`\n`) as delimiters. The resulting tokens are stored in `token_list`, and the function returns the number of tokens. Below shows an example of showing the output of `token_list` for a given `str`.

```
str = “/home/user/text.txt”
token_list =
```

“home”	“user”	“text.txt”

Hint: Use `strtok()` to tokenize the string. The last index of `token_list` will be the file name, while the rest are directory names.

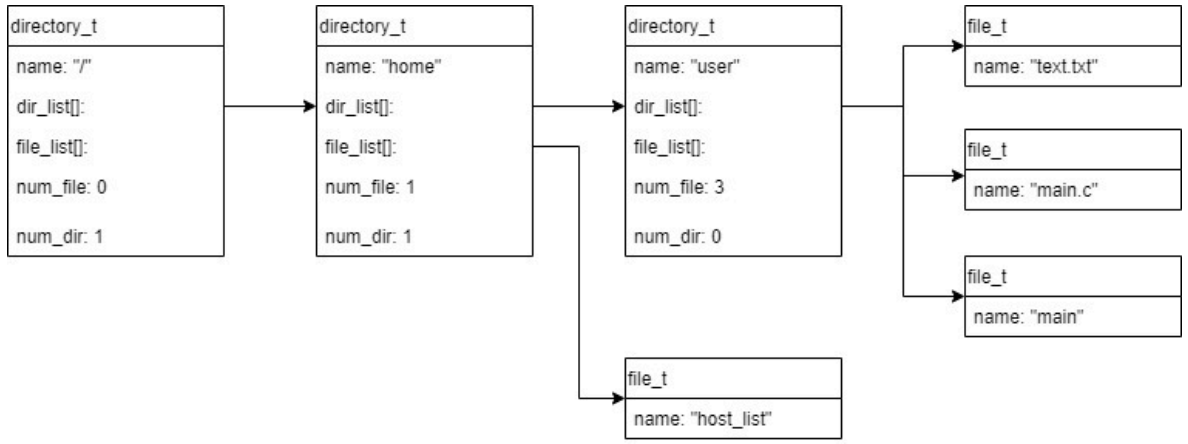
3.2 Task #2: Implement `make_dir_and_file()` in `dir_file.c`

```
 // This function creates the hierarchy of files and directories
 // which are indicated by token_list. Everything starts in root_dir.
 // You can implement this function using the above find_create_dir()
 // and find_create_file() functions.
 void make_dir_and_file(directory_t* root_dir, char** token_list, int
num_tokens) {
     /* Fill this function */
 }
```

This function constructs a hierarchical structure of directories and files starting from `root_dir`. The number of tokens in `token_list` is given by `num_tokens`. Note that the `num_token` parameter can be passed from the main function when the first task is correctly implemented.

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

<Paths for files>



<data structure in the skeleton code>

The figure above shows a diagram of the structure of directories and files for the tokens in `token_list`. The structures for `directory_t` and `file_t` are defined in `include/dir_file.h`, so you can easily implement them by referring to that section.

IMPORTANT: Do not modify any files other than utils.c and dir_file.c (e.g., main.c).

4. Getting Started

If you are unfamiliar with Linux, please read and watch the following resources:

- Linux Shell: https://missing.csail.mit.edu/2020/shell-tools/
- Vim editor: https://missing.csail.mit.edu/2020/editors/
- Makefile: https://missing.csail.mit.edu/2020/metaprogramming/

4.1 Downloading the provided skeleton code

```
$] git clone https://github.com/Sys-ku/kece343-comm307-project0.git
Cloning into 'kece343-comm307-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.
```

4.2 Compiling the source code

```
$] pwd
/home/comarch/st<Your student ID>

$] cd kece343-comm307-project0/
$] pwd
/home/comarch/st<Your student ID>/kece343-comm307-project0/
```

Let's move to the directory where you download the skeleton code

```
$] 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 ./utils.o ./main.o ./dir_file.o
```

With the `make` command, you can generate an executable file, `pa0`.

4.3 Running the executable file

```
$] ls
dir_file.c      include  main.o      my_outputs  README.md
sample_outputs  utils.o  dir_file.o  main.c      Makefile
pa0             sample_inputs   utils.c
```

You can find the executable file, `pa0`, through the `ls` command in the source directory. Below is how to execute `pa0`. For reference, the input files used in this assignment exist in the `sample_inputs` directory, and the reference output corresponding to the `sample_inputs` exist in the `sample_outputs` directory.

```
# Executing the example0 case
# ./pa0 <file_input> <dir_input>
$] ./pa0 sample_inputs/input0/file_input  sample_inputs/input0/dir_input

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

To run an executable file on the shell prompt, you can run it by adding '`./`' in front of the executable file. `pa0` has two input parameters: `file_input` and `dir_input`. Therefore, to run the program normally, you must input the arguments together like the example above.

4.3.1 Running example for `sample_inputs/input0/`

dir_input	file_input
/home/jinu/git/project0 /home/jinu/ /home	/home/jinu/git/project0/main.c /home/jinu/git/project0/README.md /home/jinu/git/project0/.gitignore /home/jinu/figure1.png /home/jinu/figure2.png

Again, `pa0` is designed to take two text files as input parameters, named `dir_input` and `file_input` respectively.

- `file_input`: a text file containing the absolute paths to the files
- `dir_input`: a text file containing the absolute paths to the directories

In the example, there are three files: `main.c`, `README.md`, and `.gitignore` in the `/home/jinu/git/project0` directory, and `figure1.png` and `figure2.png` files in the `/home/jinu` directory.

For `sample_inputs/input0/file_input` and `sample_inputs/input0/dir_input`, `pa0` shows the following output.

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

Please note that `pa0` only outputs files, not directories. Also, if a directory exists but there are no files in it, it will not show them. The output order is as follows:

1. The name of directories written in `dir_input`

2. If there are multiple files in a directory, the output should be the order written in `file_input` (not alphabetical order)

4.3.2 Testing your `pa0`

To determine if your `pa0` is generating the correct output, you can check that the contents of the files in the provided sample_outputs match the output. However, it is very cumbersome to compare them one by one by eye. Fortunately, Linux provides a utility called `diff` to help with this task. Note that `diff` only takes files as input for comparison, so in order to use `diff`, you need to save the output from `pa0` to a file and compare it. In Linux, you can use '`>`' (called redirection) to save the output to a file. In the example below, executing the first command creates a file called `my_outputs/output0` and saves the output of `pa0` to that file. Then, you can use `diff` to compare the files in `sample_outputs` with the files where the output is saved. If they match, it does not output any message, and if they do not, it outputs the incorrect part.

```
# The output of ./pa0 is saved in the file my_outputs/output0
# Usage: ./executable > file_name
$] ./pa0 sample_inputs/input0/file_input  \
> sample_inputs/input0/dir_input > my_outputs/output0

$] ls my_outputs/output0
output0

$] diff -Naur my_outputs/output0 sample_outputs/output0
```

Below is the result output via `diff` when running for the 0th test (`test_0`) in the `Makefile` provided for the assignment.

```
$] 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
```

You can use the five test cases (`test_0` ~ `test_4`) defined in the `Makefile` to compare your output with the output of the reference program to find out where your output is wrong. The '`make test`' command shows the test results for all test cases 0 through 4.

If your output result is the same as the output of the reference program, it appears as follows.

```
$] make test_0
```

```
Testing example0
      Test seems correct
```

5. Submission

5.1 Prepare for submission

After you have tested your code and it seems to be working correctly, it's time to submit it. In your project directory, simply type: `make submission`

```
$] make submission
Generating a compressed file (pa0-submission.tar.gz) including utils.c
and dir_file.c
a utils.c
a dir_file.c
A compressed archive file (pa0-submission.tar.gz) has been generated.
```

After that, a compressed archive file (`pa0-submission.tar.gz`) has been generated. If you have your code in our exercise server, you need to download it to your local computer. Use the SCP protocol or a similar TUI application to download it.

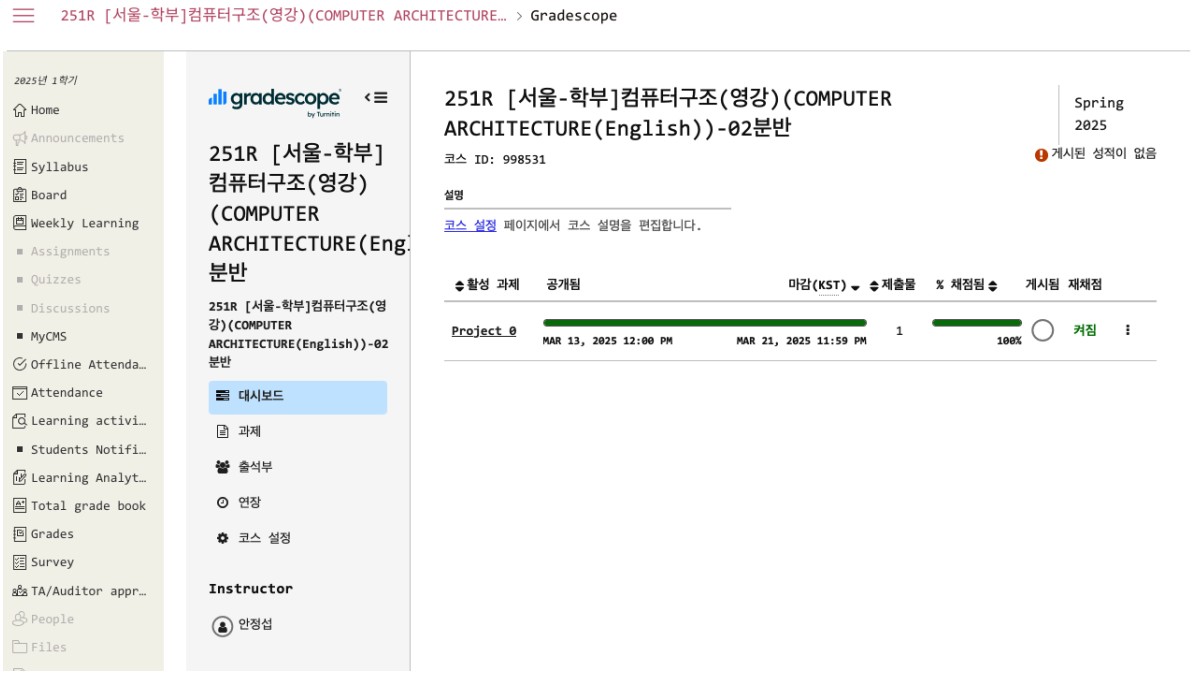
This is the file you need to submit. Now, go to the Korea University LMS and navigate to your course.

Do not change your archive file name! If you do, our system cannot check your source code.

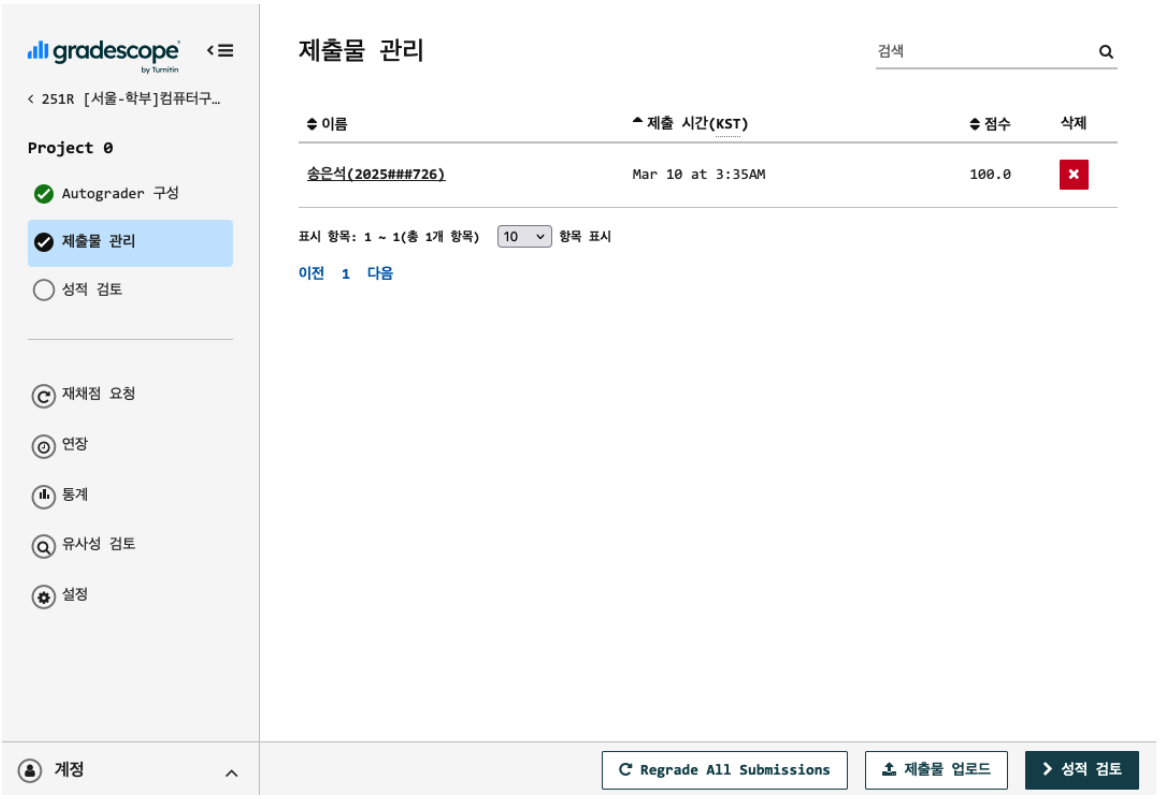
5.2 Upload your submission

1. Click the Gradescope button on your left menu.

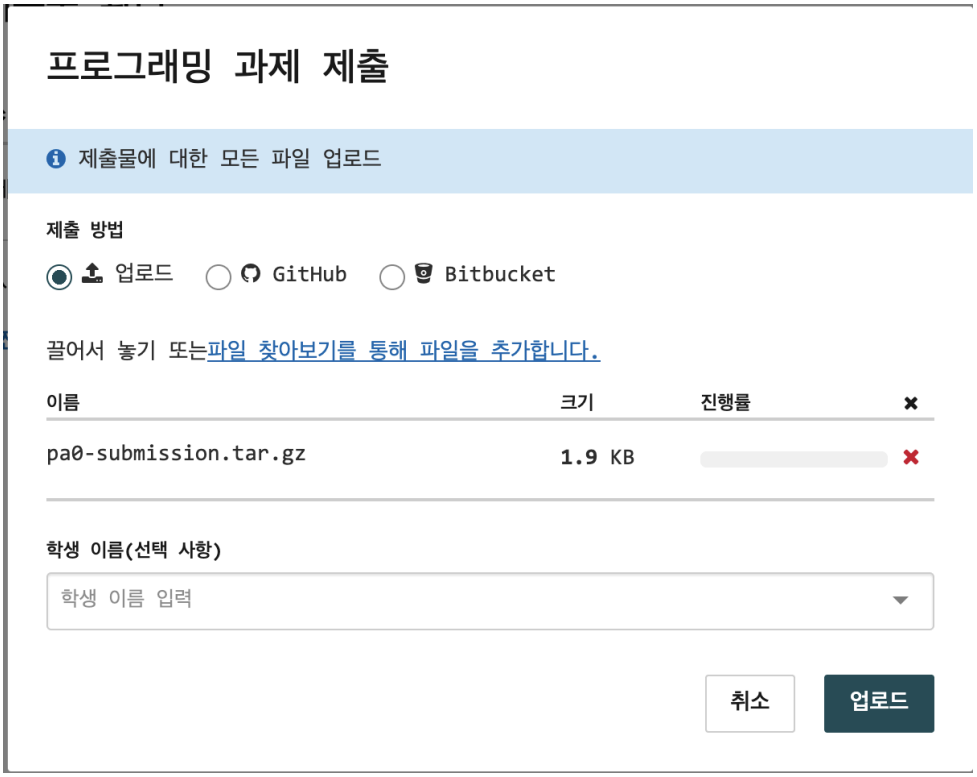
2. There is an active assignment. In this case, click Project 0.



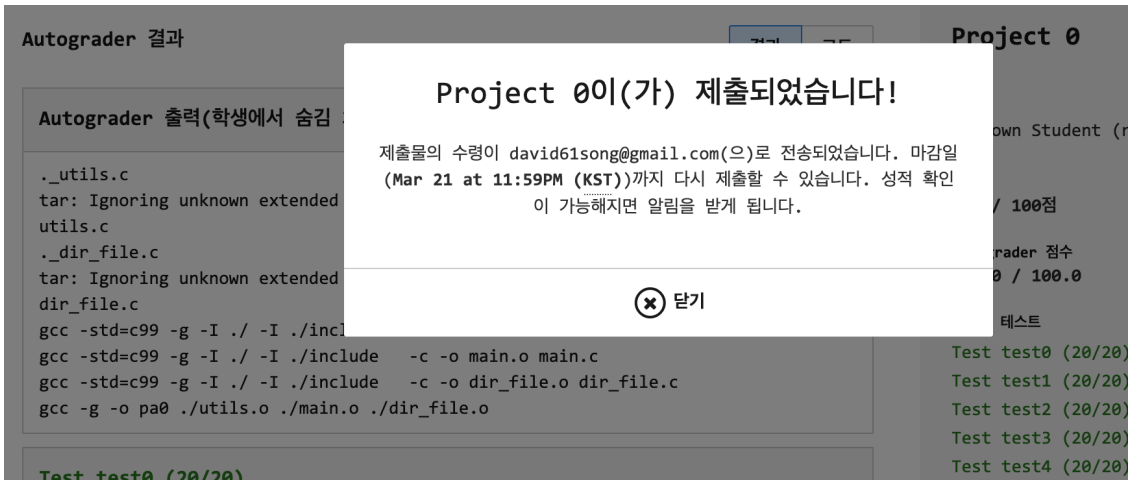
3. Click the Submit button (제출물 업로드).

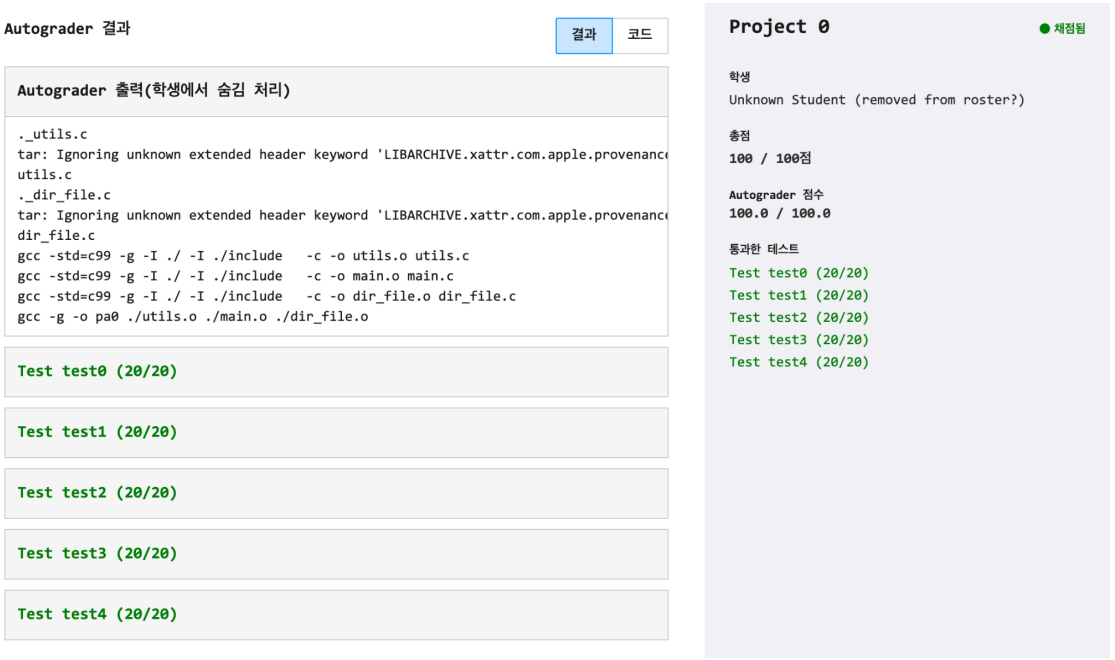


4. Upload the `pa0-submission.tar.gz` file. Do not change your archive file name! If you do, our system cannot check your source code.



5. If the upload is successful, you will receive an email with the test results of your code.





If the upload is successful, you will receive an email with the test results of your code.

5.3 Submission tips

As mentioned earlier, do not change your .tar.gz file name. Also, do not include any files other than `utils.c` and `dir_file.c` in your archive file. The `main.c` file and `Makefile` is already hardcoded in our grading server, so if you attempt to include your version of main.c or any other files, the program build will likely fail.

We suggest building your code in our exercise server (Linux system).
Our grading system uses Ubuntu 22.04, gcc 11.2.0-1ubuntu1, and make 4.3-4.1build1.

Some other Unix-like systems (like macOS, BSD, etc.) may use different versions of libc, so we strongly suggest that you check your build in Linux (Ubuntu). Build failures will cause all test cases to fail.

6. Tips (or FAQ)

6.1 Segmentation Fault?

A segmentation fault occurs when your program accesses memory incorrectly, such as:

- Using uninitialized pointers
- Accessing memory out of bounds

6.2 GDB (GNU Debugger)

GDB is a well-known debugger. It can trace the functions or lines of code that are being processed inside a program, and at the same time, check the variables or result values at that point to confirm that the program is operating normally.

6.2.1 Running GDB

The commands below are frequently used in GDB and they will be enough to perform the assignment. If you run the executable program by adding '`gdb`' in front of it, you can debug the program through GDB. If the program you want to run takes arguments, you should use the '`--args`' option with '`gdb`' (see the example below). GDB provides a prompt so that the user can control the program and operates according to the entered commands. GDB commands will be explained in more detail later, and here we will explain how to run a program in GDB and how to exit GDB. If you enter '`run`' or '`r`' in the GDB prompt, the program will be executed. Since the program is not executed when GDB first starts, you need to execute the run command additionally. If a specific error occurs in the middle, the program will stop at the part where the problem occurred, and on the contrary, if no error occurs, the program will end normally. If you want to exit GDB, you can exit normally by entering '`quit`' or '`q`' in the prompt.

```
$] pwd
/home/jinu/kece343-comm307-project0/

# gdb <executable>
$] gdb --args ./pa0 sample_inputs/input0/file_input \
> sample_inputs/input0/dir_input
(gdb) run
Starting program: /home/jinu/project/kece343-comm307-project0/pa0
sample_inputs/input0/file_input sample_inputs/input0/dir_input
main.c
README.md
.gitignore
figure1.png
figure2.png
[Inferior 1 (process 2934329) exited normally]

(gdb) r
```

```
Starting program: /home/jinu/project/kece343-comm307-project0/pa0
sample_inputs/input0/file_input sample_inputs/input0/dir_input
main.c
README.md
.gitignore
figure1.png
figure2.png
[Inferior 1 (process 2935148) exited normally]

(gdb) quit

$] # return to the shell prompt
```

6.2.2 GDB commands

- **run / r:** start or restart the program

- **break / b *function***: set a breakpoint at a function
  break *linenumber*: set a breakpoint at line *linenumber* in the current source file
  break *filename:linenumber*: set a breakpoint at line *linenumber* in source file *filename*

- **delete / d**: delete all breakpoints
  delete [Number]
  delete *breakpoint*: delete the specified breakpoint

- **continue / c**: continues program execution after a breakpoint

- **next / n**: execute the next line of code

- **list** *linenumber:* print lines centered around line number *linenum* in the current source file.
  list *function*: print lines centered around the beginning of function *function*.

- **up**: move to the caller function

- **print / p *variable***: print the value of ***variable***
  e.g. If there is an array called **`list`** and the first index value is 25, you can access it as below and print the value at index 0.

  ```
  (gdb) print list[0]
  $1 = 25
  ```

  It can also be applied to expressions as well as variables.

  ```
  (gdb) print list[0]+list[0]
  $1 = 50 # 25 + 25
  ```

6.3 Finding the cause of segmentation faults

In this subsection, we will introduce how to fix our misbehaving program, which was originally designed to cause a segmentation fault.

```
$] pwd
/home/csl/

$] gdb --args ./pa0 sample_inputs/input0/file_input \
> sample_inputs/input0/dir_input
(gdb) r
Starting program: /home/jinu/project/kece343-comm307-project0/pa0
sample_inputs/input0/file_input sample_inputs/input0/dir_input

Program received signal SIGSEGV, Segmentation fault.
__memmove_avx_unaligned_erms () at
../sysdeps/x86_64/multiarch/memmove-vec-unaligned-erms.S:374
374     ../sysdeps/x86_64/multiarch/memmove-vec-unaligned-erms.S: No
such file or directory.
```

GDB says that a segmentation fault occurs in `memmove-vec-unaligned-erms.S`. However, since the file belongs to a different library and not the source code we have, we cannot access the code, making debugging difficult. Therefore, it is more efficient to go to the place in your code that calls the file and debug it. You can use **`up`** as a command to move to the previous function.

```
(gdb) up
#1  0x00005555555558e7 in create_file (name=0x55555555d3d0 "main.c") at
dir_file.c:60
60          memcpy(file->name, name, MAX_NAME_SIZE);
```

The place where the previous error occurred is `memcpy(file->name, name, MAX_NAME_SIZE);` on line 60 of `dir_file.c`. If so, the first thing to check is the arguments used in the function. Let's find them one by one using the **`p`** command.

```
(gdb) p file
$1 = (file_t *) 0x55563810
(gdb) p file->name
Cannot access memory at address 0x55563810
```

It says that `file->name` is inaccessible. In other words, we can see that memory has not been allocated to `file->name`. Then, we exit the debugger and carefully examine the part of the source code that allocates or uses `file->name`, and modify the code. This will allow us to resolve the segmentation fault more easily than we might think.