## **2. How to open a file**

Opening and closing a file is one of the basic techniques.

Error trapping is important because files that exist outside the program may not actually exist.

It is a good way to write a program by predicting that an error will occur in advance and thinking about how to deal with it.

In C language, error handling is (unfortunately) basically done by yourself.

(Therefore, there is an advantage that you can take the advanced tactic of ignoring the error.)

In order to be able to give a file desired by a user to the program from the command line, the arguments of the main () function should be learnt here.

### **02.01. Open a file carelessly**

Use the fopen() function to open a file.

When you check fopen with man, if you want to use the fopen() function, you will be instructed to write "#include <stdio.h>" in advance, so write it at the beginning of the source.

(Actually, the printf() that you usually use already requires the same thing, so you don't need to write it twice.)

You specify the target file name and how to open it (here you want to read the file, so "r" is specified to indicate your request) on using the function.

(See also the man page for the second argument of fopen())

If the file is opened successfully, you will be given a "file tag (pointer to the FILE structure)" corresponding to the target file.

(Formally, it is called a file descriptor.)

After that, data input / output is performed using the tag.

After use, use the fclose() function to close the file.

This was not necessary when using standard I/O.

This is because the compiler always closes the standard output silently at the end of the execution.

The compiler compiles your program with such functions by secretly adding them before and after the program, you wrote so that you can perform "open for standard I/O" and "close for standard I/O" operations at the time of program execution and termination. He was kind to us.

02-01-OpenFile-NotSoGood.c (code/02-01/02-01-OpenFile-NotSoGood.c)

The main points of this new program are argc and argv, which are the arguments of the main() function.

From argc, you can see the number of arguments at runtime. This number includes the name of the executable file itself.

argv[] is an array of string pointers. This array is prepared for the size of argc (0 to argc-1).

When you refer to one character string pointer, the character string continues from there to the byte containing'\ 0'.

(By the way, how is the character '\0’ in memory? You should have learned it.)

For example, if you set

$ ./02-01-OpenFile-NotSoGood

The main function of the user program is called by the OS at the start where:

- argc is1

- argv[0] is the first-byte address of an area of 27 bytes containing the string "./02-01-OpenFile-NotSoGood"

In another example, if you set

$ ./02-01-OpenFile-NotSoGood abc.txt 2ndoption.log 2011

- argc is 4

- argv[0] is the first-byte address of an area of 27 bytes containing the string “./02-01-OpenFile-NotSoGood"

- argv[1] is the first-byte address of the 8-byte area containing the character string "abc.txt"

- argv[2] is the first byte address of the 14-byte area containing "2ndoption.log"

- argv[3] is the first byte address of the 5-byte area containing "2011" (note that this is letters, not an integer value)

(You may be aware of one extra byte for each string area. It is used to put the string terminator '\ 0')

This program works, but sometimes it gets stucked.

Let's actually try it.

|  |
| --- |
| $ gcc -Wall -o 02-01-OpenFile-NotSoGood 02-01-OpenFile-NotSoGood.c  $ emacs abc.txt (Let's create something. This time, the contents can be empty.)  $ ls (02-01-OpenFile-NotSoGood, 02-01-OpenFile-NotSoGood.c, abc.txt should be there)  $ ./02-01-OpenFile-NotSoGood abc.txt (works properly)  $ ./02- 01-OpenFile-NotSoGood (falls)  $ ./02-01-OpenFile-NotSoGood def.txt (falls)  $ chmod u-r abc.txt (If you yourself cannot read abc.txt ...)  $. / 02-01-OpenFile-NotSoGood abc.txt (It will fail even if you specify existing abc.txt)  $ chmod u+r abc.txt (If you restore abc.txt so that you can read it yourself ...)  $ ./02-01-OpenFile-NotSoGood abc.txt (returned to work properly) |

How many segmentation faults have you experienced? Do you understand the reasons?

All of these segfaults can be avoided by writing a well-behaved program (writing the correct and beautiful C language documents).

You should avoid these critical errors.

**Exercise**

**02-01-ex1**: Describe the characteristics of functional programming languages ​​in a manner that is fit to the explanation in this section.

**02-01-ex2** : Explain the general usage of the UNIX command chmod in a way that matches the use in this section.

**02-01-ex3**: What happens if you write "#include <stdio.h>" more than once in the program?

(Hint: Answers could be found in section 04.02.)

### **02.02. Open files elegantly**

The program below is a program that can handle various situations even if the number of files to be set as arguments.

02-02-OpenFile-Full.c (code/02-02/02-02-OpenFile-Full.c)

Difference from 02-01 (code/02-02/02-01\_02-02.html)

A user may specify non-existing files.

In the above example that works properly, yet if you specify non-existent files, it will go segfault.

(If you're familiar with it, explain to the people around you who are causing segfault. If you can answer it, you are a wise man.)

(If there are no people around you, you can use either a doll or a stuffed animal.)

(An extension of this approach, is known as a teddy bear debugging. Cute.)

(Those who can't answer it are okay. I will explain it in 02.03.)

You should take action to prevent trouble.

In fact, for most of the cases, the library functions you use will tell you in return whether they are working (or working) as expected.

You can use that value well.

For new library functions that will appear in the future, please refer to the man page for knowing their usage and the meaning of the return value.

(I don't prepare man pages from now on like before.)

(You can google. However, if you should find the correct page of Unix, not of Windows, otherwise it will be confusing. Google teacher is not perfect.)

As a computer programmer's unwritten rule, **if the return integer value is 0, it is interpreted as "as expected; peaceful"**.

In the future, when you create a function by yourself in C language and return the state as a return value, make it an integer type and return 0 to tell the normal end, and return a non-zero value for other abnormal ends.

(Because this is a "cultural practice", not a few programmers don't have this habit.)

(If you follow it, wise people will think that you are cool. See the exercise for the reason. )

Also, get in the habit of closing the file with fclose() as soon as you run out of use.

(Organization will help you in the future, avoiding bugs!)

**Exercise**

**02-02-ex1**: In a C program written by a familiar programmer, 0 is set for a function that returns an integer at the end of a normal operation. In many cases, -1 is returned at the time of abnormal termination.

- There is a reason for this. Look it up and try to explain it.

(Hint: 2's complement expression)

**02-02-ex2**: When you create an executable file from a C source file and execute it, there is an upper limit to the number of files that can be opened at the same time.

- Please express the meaning of "at the same time" accurately.

- Show the maximum number along with the survey method.

(Once upon a time, I actually got hurt by this … The number at that time was surprisingly small …)

### **02.03. Cooperation with bash**

Why does the main() function return an integer value?

That's because it's useful when calling a program from a terminal (strictly speaking, a shell called bash running there).

In the UNIX computer used in this class, a shell called bash is used in the terminal, and it acts as an intermediary between the programs and your character input / output.

(The typed characters are passed to a program, and the character strings from the program are displayed on the screen.)

Actually, all programs and commands run on the bash command line return integer values. There is an agreement to have it.

(Since bash is also a UNIX command, you can check its function on the man page)

The program created in section 02.02. returns 0 if it ends normally, and returns some other value if it ends abnormally.

On the bash side, there is an operator called a control operator that can change the behavior of bash by looking at the program return value at the end of execution.

That is "&&" and "||".

"&&" executes the next command only when the previous command returns 0, and "||" executes the next command only when the previous command returns a non-zero.

If it is ";", the next command will be executed regardless of the result of the previous command, so if you combine these operators, you will be able to do a lot of work.

(The explanation is written in the section called “Lists” of man bash.)

02-03-OpenFile-AndGo.bash (code/02-03/02-03-OpenFile-AndGo.bash)

Try typing the commands written in the above file line by line on the command line.

(As a premise, the execution program name is 02-02-OpenFile-Full)

(You may think of running all the commands together. Actually, you can do it by "$ bash ./02-03-OpenFile-AndGo.bash".)

(Such a file is called a "script". It's for bash, so it's called a "bash script".)

(In a bash script, # indicates the beginning of a comment)

(In C, comments are // or / \* \* /.)

**Exercise**

**02-03-ex1**: Create a program that displays the number of arguments and their contents.

- Display each argument with a line break.

- You do not need to handle multibyte characters.

**02-03-ex2**: Show how to type on the command line to give a string containing spaces as one of the command line arguments.

**02-03-ex3**: Answer the exact type of the variable argv.

**02-03-ex4**: Show the memory structure starting from argv with the command name "./example", the first argument "234.545", and the second argument "input.log". Show also the argc value too.