File Input/Output

Outline

1. Introduction
   - Opening a file
   - Closing a file

2. Example 1: Reading data from a file
3. Example 2: Check if a file is opened successfully
4. Example 3, 4: Reading numbers using `fscanf()`
5. Example 5: Writing data to a file
6. Example 6: Handling multiple files in a program

1. File I/O (Introduction)

- Reading data from a file is similar to reading data from a keyboard
  - Characters are read from the "input source" sequentially.
    - Keyboard input: May have to wait for additional input
    - File Input: All data are in the file.
- Writing data to a file is similar to outputting data to the console.
  - Characters are output sequentially.

1.1. File I/O is similar to Console I/O

Input

A program retrieves the data from the memory sequentially (e.g., using scanf())

Data in memory

Console Input: System reads characters from keyboard

File Input: System reads characters from a file

Output

Console output: System displays the characters on the screen

File Output: System writes the characters to a file

Data in memory

A program writes data sequentially to memory (e.g., using printf())

1.2. Steps involving in File I/O

1. ***Open*** a file – Request the system to "prepare" a file for reading/writing
2. Reading/writing data from/to the file
3. ***Close*** the file when the program is done with the file I/O
   - Closing a file means
     - Releasing all the resources in the memory associated with the file
     - Flush all data in the memory to the output file if necessary

- Step 2 is the same as performing console I/O.

Example #1: Reading data from a file

```c
#include <stdio.h>

int main(void) {
    FILE *fptr;
    int num1, num2;

    fptr = fopen("data.txt", "r");  // open file

    // Read two integers from the file
    fscanf(fptr, "%d%d", &num1, &num2);
    printf("%d %d\n", num1, num2);

    fclose(fptr);  // close file
    return 0;
}
```

Content of "data.txt"

```
123
456
```

Output

```
123 456
```

Example #1: Reading data from a file

```
1  #include <stdio.h>
2
3  int main(void) {
4      FILE *fptr;
5      int num1, num2;
6
7      fptr = fopen("data.txt",
8
9      // Read two integers from
10     fscanf(fptr, "%d%d", &num
11     printf("%d %d\n", num1, n
12
13     fclose(fptr);  // close file
14     return 0;
15 }
```

Declare a variable , `fptr`, to store the "states" of a file in the memory.

Note that the variable must be prefixed with a '*' in the declaration.

`FILE` is a data type defined in "`stdio.h`".

Example #1: Reading data from a file

```
1  #include <stdio.h>
2
3  int main(void) {
4      FILE *fptr;
5      int num1, num2;
6
7      fptr = fopen("data.txt", "r");  // open file
8
9      // Read two intege
10     fscanf(fptr, "%d%d
11     printf("%d %d\n",
12
13     fclose(fptr);  //
14     return 0;
15 }
```

Open a file named "data.txt" for reading.

The 2nd argument, "r", indicates that we are opening the file for reading only.

Example #1: Reading data from a file

```
1  #include <stdio.h>
2
3  int main(void) {
4      FILE *fptr;
5      int num1, num2;
6
7      fptr = fopen("da
8
9      // Read two integers from the file
10     fscanf(fptr, "%d%d", &num1, &num2);
11     printf("%d %d\n", num1, num2);
12
13     fclose(fptr);  // close file
14     return 0;
15 }
```

After a file has been opened successfully, we can start reading data from the file via `fptr`.

`fscanf()` behaves like `scanf()` except that `fscanf()` requires a "file pointer variable" as its first argument.

Example #1: Reading data from a file

```
1  #include <stdio.h>
2
3  int main(void) {
4      FILE *fptr;
5      int num1, num2;
6
7      fptr = fopen("data.txt", "r");  // open file
8
9      // Read two integers from the file
10     fscanf(fptr, "%d%d", &num1, &num2);
11     printf("%d %d\n", num1, num2);
12
13     fclose(fptr);  //
14     return 0;
15 }
```

Tell the system to release the resources associated with the file. e.g., free up memory space.

Example #2: Checking if a file is opened successfully

```
1  #include <stdio.h>
2  #include <stdlib.h>      // Need this in order to use exit()
3
4  int main(void) {
5      FILE *fptr;
6
7      fptr = fopen("data.txt", "r");  // open file
8      if (fptr == NULL) {
9          printf("Cannot open file!\n");
10         exit(1); // Terminate the program immediately
11     }
12     …
13     fclose(fptr);  // close file
14     return 0;
15 }
```

When fopen() fails to open a file, it returns NULL (a predefined named constant).

Example #2: Checking if a file is opened successfully

```
1  #include <stdio.h>
2  #include <stdlib.h>     // Need this in order to use exit()
3
4  int main(void) {
5      FILE *fptr;
6
7      fptr = fopen("data.txt", "r"); // Open file
8      if (fptr == NULL) {
9          printf("Cannot open file!\n");
10         exit(1); // Terminate the program immediately
11     }
12     …
13     fclose(fptr);
14     return 0;
15 }
```

Usually, when a program cannot open a file, there is not much we can do subsequently. A common practice is to output an error message and terminate the program using `exit()`.

The argument passed to `exit()` will be returned to the system. The value has the same meaning as the returned value of `main()`. Typically, a zero means the program terminates normally and a non-zero value means the program terminates abnormally.

fscanf()

```
int fscanf(FILE *fp, char *format, arg1, ...);
```

- Similar to `scanf()`, except that the data comes from the stream `fp` (instead of console input).

- On success, the function returns the number of items of the argument list successfully filled.
- If no value could be read due to either a file error or when end-of-file is reached, the function returns the named constant `EOF`.

Example #3: Reading all integers from a file

```
123
456
333
666
999
…
```

- **Objective**: To read the data from a file in the above format. i.e., each line contains an integer.

- Suppose the file is named "data.txt" and its format is valid.

Example #3: Reading all integers from a file

```
1   int main(void) {
2       FILE *fptr;
3       int num;
4
5       fptr = fopen("data.txt", "r");  // open file
6
7       while (1) {
8           if (fscanf(fptr, "%d", &num) != 1)
9               break;
10          …  // Process num here
11      }
12
13      fclose(fptr);  // close file
14      return 0;
15  }
```

We expect `fscanf()` to read one integer. If it cannot read any integer, that means we have read all the integers in the file.

Example #4: Reading all numbers from a file

```
123   1.0   2.0
456   3.5   4.4
333   2.6   7.8
666   7.7   3.32
999   9.2   5.2
…
```

- **Objective**: To read the data from a file in the above format. i.e., each line contains an integer and two floating point numbers that are separated by white space characters.

- Suppose the file is named "data.txt" and its format is valid.

Example #4: Reading all numbers from a file

```
1   int main(void) {
2       FILE *fptr;
3       int num1;
4       double num2, num3;
5
6       fptr = fopen("data.txt", "r");
7
8       while (1) {
9         if (fscanf(fptr "%d%lf%lf", &num1, &num2, &num3) != 3)
10              break;
11          …  // Process num1, num2, num3 here
12      }
13      fclose(fptr);  // close file
14      return 0;
15  }
```

We expect `fscanf()` to read three values. If it cannot read exactly three values, that means we have read all the data in the file.

Example #5: Writing data to a file

```c
#include <stdio.h>

int main(void) {
    FILE *fptr;
    int i;

    fptr = fopen("data.txt", "w");

    fprintf(fptr, "Hello!\n");
    for (i = 0; i < 10; i++)
        fprintf(fptr, "%d ", i);
    fprintf(fptr, "\n");

    fclose(fptr);
    return 0;
}
```

Content of "data.txt"

```
Hello!
0 1 2 3 4 5 6 7 8 9
```

Example #5: Writing data to a file

```
1  #include <stdio.h>
2
3  int main(void) {
4      FILE *fptr;
5      int i;
6
7      fptr = fopen("data.txt", "w");
8
9      fprintf(fptr, "Hello!\n"
10     for (i = 0; i < 10; i++)
11         fprintf(fptr, "%d ",
12     fprintf(fptr, "\n");
13
14     fclose(fptr);
15     return 0;
16 }
```

The 2nd argument, "w", indicates that we are opening the file for writing only.

If the file does not exist, `fopen()` will create a new file.

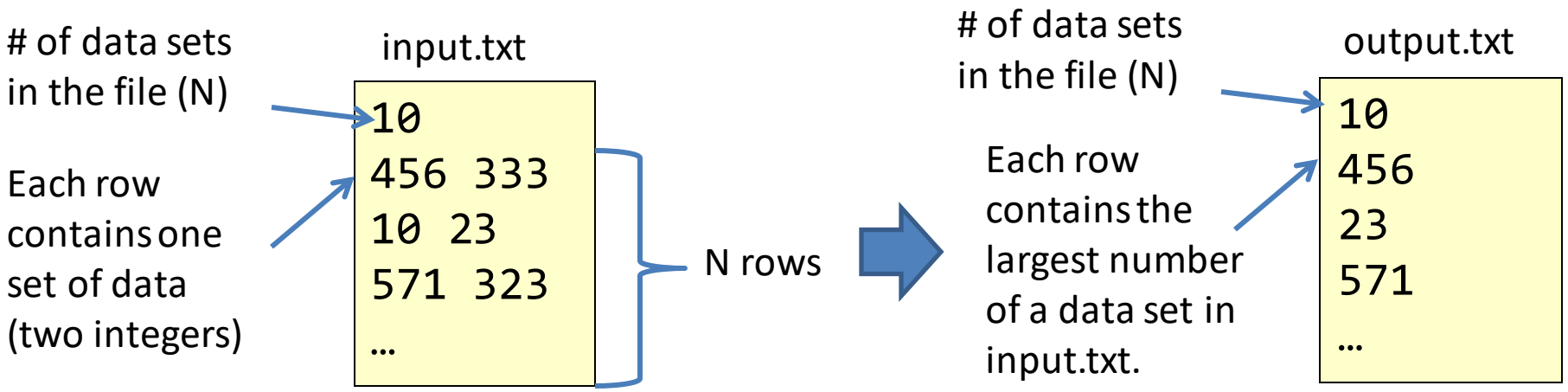
If the file already exists, `fopen()` will replace the existing file.

Example #5: Writing data to a file

```
1  #include <stdio.h>
2
3  int main(void) {
4      FILE *fptr;
5      int i;
6
7      fptr = fopen("data.txt", "w");
8
9      fprintf(fptr, "Hello!\n");
10     for (i = 0; i < 10; i++)
11         fprintf(fptr, "%d ", i);
12     fprintf(fptr, "\n");
13
14     fclose(fptr);
15     return 0;
16 }
```

`fprintf` behaves like `printf()` except that `fprintf()` requires a "file pointer variable" as its first argument.

Example #6: Dealing with multiple files in a program.



- **Objective**: Read the data from "input.txt" and output the result as shown above to "output.txt".

- We assume the input file format is valid.

Example #6: Dealing with multiple files in a program.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5      FILE *fin, *fout;   // For input and output streams
6      int i, N, num1, num2, max;
7
8      fin = fopen("input.txt", "r");
9      fout = fopen("output.txt", "w");
10
11     if (fin == NULL || fout == NULL) {
12         printf("Failed to open input or output file.\n");
13         exit(1);
14     }
15
16
```

Example #6: Dealing with multiple files in a program.

```
17      // Read # of data sets from the input file and
18      // write the number to the output file
19      fscanf(fin, "%d", &N);
20      fprintf(fout, "%d\n", N);
21
22      for (i = 0; i < N; i++) {
23          fscanf(fin, "%d%d", &num1, &num2);
24          if (num1 > num2)
25              max = num1;
26          else
27              max = num2;
28          fprintf(fout, "%d\n", max);
29      }
30      fclose(fin);
31      fclose(fout);
32      return 0;
33  }
```

Summary

- Steps for performing File I/O
  - Open a file using fopen()
  - Perform I/O through a file stream
  - When done, close the file
- How to check if a file is opened successfully
- Using `fscanf()` and `fprintf()` to read/write numbers.
- How to check if `fscanf()` is reading enough data from the file

Reading Assignment

- C: How to Program, 8th ed, Deitel and Deitel
- Chapter 11 C File Processing
  - Sections 11.1 – 11.2: File Concepts
  - Sections 11.3: Basic File Operations

Appendix: Special streams: `stdin` and `stdout`

- The special streams `stdin` and `stdout` are predefined and pre-opened in a C program for standard (console) I/O.
  - `fprintf(stdout, …)` is equivalent to `printf(…)`
  - `fscanf(stdin, …)` is equivalent to `scanf(…)`

Appendix: About filenames …

- On Windows, the file extension of known file types is hidden by default. A file named "input.txt" may only shown in the "Windows Explorer" as "input".

- On Windows, when you create a sample data file using Notepad, Notepad will automatically append the ".txt" extension to your file; you do not need to enter ".txt" when giving a file name in Notepad.

- If your Visual Studio project name is "ConsoleApplicaton1", and the project folder is "L:\ConsoleApplication1", then when you execute your program in Visual Studio, the default "working folder" of your program is "L:\ConsoleApplication1\ConsoleApplication1" (i.e., the folder where the input and output files should be located.)

Appendix: fopen()

```
FILE * fopen(char *filename, char *mode);
```

- Open the file with the name specified in **filename**.
- **mode** specifies how the file should be opened
  - If mode is **"r"**, the file is opened for input only.
  - If mode is **"w"**, the file is opened for output only. With this mode, fopen() will always create a new file. If a file with the same name exists, the file will be replaced.
  - There are other modes, but those are less common modes.

- The function returns a non-NULL pointer (of type FILE *, usually referred to as a *stream*) if the file can be opened successfully. Otherwise the function returns NULL.

Appendix: fclose()

```
int fclose(FILE *fp);
```

- Releases the "resource" associated the a file stream.

- If the file stream is an output stream, then the function also causes the buffered data associated with the stream `fp` to be written to the disk.

- Not closing a stream after use may cause loss of data.