Chapter 13

File Input/Output & Command Line Arguments

Dr. Niroj Kumar Pani
nirojpani@gmail.com

Department of Computer Science Engineering & Applications
Indira Gandhi Institute of Technology
Sarang, Odisha

Chapter Outline…

- File Input/Output
  - Background
  - Files
  - File I/O Concept
  - Opening a File
  - Closing a File
  - Performing I/O Operations on Files
  - Error Handling During I/O Operations
  - Programming Example
- Command Line Arguments
  - What are They & How are They Used?
  - Programming Examples
- Assignments

File Input/Output

Background

- **Limitations of Console I/O:**
  - Until now we have been using the console I/O functions such as `printf()`, `scanf()` to read and write data.
  - The console I/O functions use the console / terminal, i.e., the keyboard and the monitor as their target place.
  - This works fine as long as the data is small. However, many real-life applications deal with large volumes of data. In such situation, the console I/O approach becomes very inefficient for two reasons.
    - It is time consuming to enter (and re-enter) input data through the keyboard.
    - The input and output data is lost (can't be stored) every time when the program is terminated, or the computer is turned off.
- **The Solution:** We need a place where we can store data (for reading or writing) permanently… and that's a file.

Files

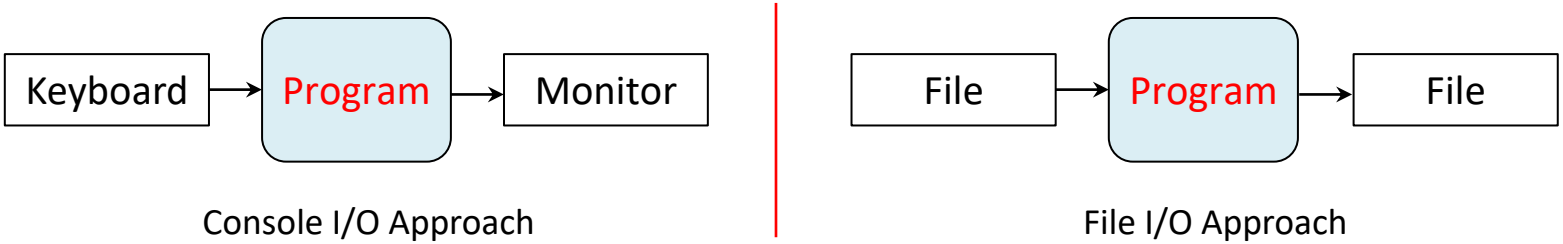
- **What is a File?: A named collection of data, stored in secondary storage.**
- **How is a file stored?**
  - **Stored as a sequence of bytes, logically contiguous (may not be physically contiguous).**
  - **The last byte of a file contains the** end-of-file character (EOF)**, with ASCII code** 1A **(hex).**
- **Types of Files: Two types**
  - **Text: Contains ASCII codes only.**
  - **Binary: Can contain non-ASCII characters** (image, audio, video, executable).

[NOTE]: In C, the console (the standard I/O devices, i.e., the keyboard and the monitor) can also be represented as files. Three special files called streams are defined in **`<stdio.h>`**.

- **`stdin:`** represents the standard input device (the keyboard).
- **`stdout:`** represents the standard output device (the monitor).
- **`stderr:`** represents the standard error display device (usually also the monitor).

File I/O Concept

- **Console I/O Vs. File I/O:**



- **Steps in File I/O:** **File I/O involves the following steps (in order)**
  1. **Opening a file**
  2. **Performing the input / output operations**
  3. **Closing the file**

These steps are carried out through some standard file I/O functions provided by C. We see some important file I/O functions first. Then we discuss these steps in detail.

- **Standard File I/O Functions:** **C language has provided several high-level functions for file I/O. Some of the important functions are listed below:**

Function name	Operation
`fopen()`	Opens an existing file. If the file doesn't exist, it creates the file and then opens it.
`fclose()`	Closes an opened file.
`getc()`	Reads a character from a file.
`putc()`	Writes a character to a file.
`getw()`	Reads an integer from a file.
`putw()`	Writes an integer to a file.
`fscanf()`	Reads a set of values from a file.
`fprintf()`	Writes a set of values to a file.

[Table 13.1: Standard File I/O Functions]

Opening a File

- **This is the 1st step while working with files. We must open a file before using it.**
- **Syntax:**

```
FILE *file_pointer;
file_pointer = fopen("file_name", "mode");
```

e.g.,

```
FILE *fp;
fp = fopen("Data.txt", "w");
```

Explanations:

- **In C, every file belongs to a data type** FILE **which is a pre-defined data structure in the I/O library. Therefore, all files must be declared as type** FILE **before using them. So, in C, the standard syntax to open a file is:**
  1) **Declare a pointer of type** FILE (the 1st statement).
  2) **Open a file using** fopen() **and assign it to the file pointer** (the 2nd statement)

  In the remaining part of the program , the file is accessed not by its name "Data.txt" **but through the file pointer** "fp".

- **The function `fopen()` takes two strings** (should be enclosed in double quotes) **as its arguments.**
  - **The first is a "file name"** (any valid file name in the operating system), **e.g.,** "Data.txt"**. It may contain two parts, a primary name** (e.g., data) **and an optional period with an extension** (e.g., .txt).
  - **The second is the "mode"** (opening mode)**. It can be one of the following:**
    - **"r": opens the file for reading only.**
    - **"w": opens the file for writing only** (writes from beginning of the file over previous content, if any. So, be careful)**.**
    - **"a": opens the file for appending** (writing at the end of the file).

  [NOTE]: We can add a "b" character for a binary file.

  Example: `fp = fopen("Xyz.jpg", "r");`

- **Working:**
  - **The function `fopen()` opens an existing file specified by the file name** (the file should be present in the same folder where the source code is present).
  - **If the file doesn't exist, it first creates a file with the file name** (in the same folder where the source code is present) **and then opens it as per the specified mode.**
  - **If it fails to open / create the file it returns a** NULL **to** fp.

Closing a File

- This is the last step while working with files. A file must be closed after completion of all I/O operations.
  - This ensures that all the outstanding information associated with the file is flushed out from the buffers and all the links to the file are broken.
  - It also prevents any accidental misuse of the file.
- **Syntax:**

```
fclose(file_pointer);
```

e.g.,

```
FILE *fp1, *fp2;
fp1 = fopen("Avg.c", "r");
fp2 = fopen("Result.txt", "a");
...
fclose(fp1);
fclose(fp2);
```

- **Working:** It closes the file associated with the file pointer.

Performing I/O Operations on Files

- Once a file is opened, I/O operations on it can be performed by using the standard I/O functions as listed in Table 13.1.

Reading & Writing a Character: `getc()` & `putc()`

- `getc()` **&** `putc()` **are analogous to** `getchar()` **&** `putchar()` **respectively.**
- **Syntax & Working of** `getc()` **[In comparison with** `getchar()`**]:**

```
character_variable = getchar();
/* Working: It reads a character typed from the keyboard and stores it in "character_variable". */

character_variable = getc(file_pointer);
/* The only difference in the syntax is the inclusion of "file_pointer" as the argument. */
/* Working: It reads a character from a file opened in read mode whose file pointer is
"file_pointer" and stores the character in "character_variable". It reads the character from the
beginning of the file and moves to its next position for each of its call (till the file is kept open).
When the end of file is reached, it returns EOF. Thus, reading should be stopped at EOF. */
```

e.g.,

```
char c1, c2, c3;
FILE *fp;
...
c1 = getchar();  /* Reads a character from the keyboard & stores it in "c1". */
...
fp = fopen("Data.txt", "r");
c2 = getc(fp);  /* Reads a character from the file "Data.txt" & stores it in "c2". */
...
c3 = getc(stdin);  /* Reads a character from "stdin" (i.e., the keyboard) and
                      stores it in "c3". So, same as writing getchar(). */
```

■ Syntax & Working of `putc()` [In comparison with `putchar()`]:

```
putchar(character_variable_or_constant);
/* Working: It writes the character contained in "char_variable_or_constant" to the screen. */

putc(character_variable_or_constant, file_pointer);
/* The only difference in the syntax is the inclusion of "file_pointer" as the 2nd argument. */
/* Working: It writes the character contained in "character_variable_or_constant" to a file opened
in write mode whose file pointer is "file_pointer". It writes the character from the beginning of
the file and moves to its next position for each of its call (till the file is kept open). */
```

e.g.,

```
char c1, c2, c3;
FILE *fp;
...
putchar(c1);  /* Writes the character contained in "c1" to the screen. */
...
fp = fopen("Data.txt", "w");
putc(c2, fp);  /* Writes the character contained in "c2" to the file "Data.txt". */
...
putc(c3, stdout);  /* Writes the character contained in "c3" to "stdout" (i.e., the
                      screen). So, same as writing putchar(). */
```

Reading & Writing an Integer: `getw()` & `putw()`

- `getw()` **and** `putw()` **are exactly same as** `getc()` **and** `putc()` (both syntax-wise and working-wise)**. The only difference is that** `getw()` **and** `putw()` **deal with single integer while** `getc()` **and** `putc()` **deal with single character.**

- **Syntax & Working of** `getw()`**:**

```
integer_variable = getw(file_pointer);
/* Working: It reads an integer from a file opened in read mode whose file pointer is
"file_pointer" and stores the integer in "integer_variable". It reads the integer from the beginning
of the file and moves to its next position for each of its call (till the file is kept open). When the
end of file is reached, it returns EOF. Thus, reading should be stopped at EOF. */
```

e.g.,

```
int x, y;
FILE *fp;
...
fp = fopen("Data.txt", "r");
x = getw(fp);  /* Reads an integer from the file "Data.txt" and stores it in "x". */
...
y = getw(stdin);  /* Reads an integer from "stdin" (i.e., the keyboard) and
                     stores it in "y". */
```

Syntax & Working of `putw()`:

```
putw(integer_variable_or_constant, file_pointer);
/* Working: It writes the integer contained in "integer_variable_or_constant" to a file opened in write mode whose file pointer is "file_pointer". It writes the integer from the beginning of the file and moves to its next position for each of its call (till the file is kept open). */
```

e.g.,

```
int x, y;
FILE *fp;
...
fp = fopen("Data.txt", "w");
putw(x, fp); /* Writes the integer contained in "x" to the file "Data.txt". */
...
putw(y, stdout); /* Writes the integer contained in "y" to "stdout" (i.e., the
                    screen). */
```

Reading & Writing Any Number & Types of Values: `fscanf()` & `fprintf()`

- `fscanf()` & `fprintf()` are analogous to `scanf()` & `printf()` respectively.
- Since, `fscanf()` and `fprintf()` are used to read/write any number and type of data they are mostly used to perform file I/O.
- Syntax & Working of `fscanf()` [In comparison with `scanf()`]:

```
scanf("format_string", address_list);
/* Working: It reads the values typed from the keyboard in accordance with the "format
specifications" given in the "fromat_string" and stores them at addresses as given by the
"address_list" in order.  */

fscanf(file_pointer, "format_string", address_list);
/* The only difference in the syntax is the inclusion of "file_pointer" as the 1st argument. */
/* Working: It reads the values from a file opened in read mode whose file pointer is
"file_pointer" in accordance with the "format specifications" given in the "fromat_string" and
stores them at addresses as given by the "address_list" in order. It reads the values from the
beginning of the file and moves to its next position for each of its call (till the file is kept open).
Like scanf(), it also returns the number of items that are successfully read. When the end of file is
reached, it returns EOF. */
```

e.g.,

```
int rollNo1, rollNo2, rollNo3;
char name1[20], name2[20], name3[20];
FILE *fp;
...
scanf("%d %s", &rollNo1, name1);
/* Reads an integer and a string from the keyboard and stores them at addresses of
"rollNo1" and "name1" respectively. */
...
fp = fopen("Data.txt", "r");
fscanf(fp, "%d %s", &rollNo2, name2);
/* Reads an integer and a string from the file "Data.txt" and stores them at address of
"rollNo2" and "name2" respectively. */
...
fscanf(stdin, "%d %s", &rollNo3, name3);
/* Reads an integer and a string from "stdin" (i.e., the keyboard and stores them at
address of "rollNo3" and "name2" respectively. So, same as writing scanf(). */
```

- **Syntax & Working of `fprintf()` [In comparison with `printf()`]:**

```
printf("format_string", variable_or_constant_list);
/* Working: It writes the values of the variables/constants in "variable_or_constant_list" to the screen in accordance with the "format specifications" given in the "format_string". */

fscanf(file_pointer, "format_string", variable_or_constant_list);
/* The only difference in the syntax is the inclusion of "file_pointer" as the 1st argument. */
/* Working: It writes the values of the variables/constants in "variable_or_constant_list" to a file opened in write mode whose file pointer is "file_pointer" in accordance with the "format specifications" given in the "fromat_string". It writes the values from the beginning of the file and moves to its next position for each of its call (till the file is kept opened). Like printf(), it also returns the number of items that are successfully written. */
```

e.g.,

```
int rollNo1, rollNo2, rollNo3;
char name1[20], name2[20], name3[20];
FILE *fp;
...
printf("%d %s", rollNo1, name1);
/* Writes the integer contained in "rollNo1" and the string contained in "name1" to the screen. */
...
fp = fopen("Data.txt", "w");
fprintf(fp, "%d %s", rollNo2, name2);
/* Writes the integer contained in "rollNo2" and the string contained in "name2" to the file "Data.txt". */
...
fscanf(stdin, "%d %s", &rollNo3, name3);
/* Writes the integer contained in "rollNo3" and the string contained in "name3" to "stdout" (i.e., the screen). So, same as writing printf(). */
```

Error Handling During I/O Operations

- It is possible that an error may occur during I/O operations on a file.
- Typical error situations include:
  - Device overflow.
  - Trying to use a file that has not been opened.
  - Trying to open a file with an invalid file name.
  - Trying to perform an operation on a file, when it's opened in another mode.
  - Trying to read beyond EOF.
  - Trying to write to a file which is write-protected.
- If we fail to check such I/O errors, a program may behave abnormally (e.g., may terminate prematurely, may give incorrect output etc.) when an error occurs.
- The deal with such situation, C has provided two status-inquiry file handling library functions: `fefo()` and `ferror()` that can help to detect I/O errors in files.

Syntax & Working of `feof()`:

```
feof(file_pointer);
/* Working: It returns a non-zero integer if the end of file has been reached, i.e., if all the data
from the file has been read. It returns zero otherwise. */
```

e.g.,

```
int data;
FILE *fp;
...
fp = fopen("Data.txt", "r");

if(feof(fp))
{
    printf("End of file, can't read. ");
    exit(0);
}
fscanf(fp, "%d", &data);
```

- **Syntax & Working of `ferror()`:**

```
ferror(file_pointer);
/* Working: It returns a non-zero integer if an error (any error) has been detected up to that point, during processing. It returns zero otherwise. */
```

e.g.,

```
int data;
FILE *fp;
...
fp = fopen("Data.txt", "w");

if(ferror(fp))
{
    printf("Some error occurred, not able to write. ");
    exit(0);
}
fprintf(fp, "%d", data);
```

- [NOTE]: Before writing / appending we should use `ferror()` for detecting the presence of any error in the file, while before reading we should use both `feof()` and `ferror()` to make sure that the end of file has not reached and there is no error in the file.

Programming Example

```
/* PR13_1.c: Write a program that reads the "registration number", "name", and "CGPA" of three students
and writes them in a file “StudentsData.txt”, and again reads the same data from the file “StudentsData.txt”
and displays it on the screen. */

# include <stdio.h>
# include <conio.h>

struct Student
{
    int regdNo;
    char name[30];
    float cgpa;
};

void main()
{
    int i;
    char c;
    struct Student studs[3];
    FILE *fp;

    printf("\nEnter Students Information (for 3 Students)\n");
    printf("=========================================\n");
```

[Cont.]

```
/* Reading data from keyboard and storing them in variables. */
for (i=0;i<3;i++)
{
    printf("\nStudent %d\n----------\n", i+1);
    printf("Regd. No.: ");
    scanf("%d", &studs[i].regdNo);
    printf("Name: ");
    scanf("%s", studs[i].name);
    printf("CGPA: ");
    scanf("%f", &studs[i].cgpa);
}

/* Initiate creating/opening the file "StudentsData.txt" in write mode. */
fp = fopen("StudentsData.txt","w");
if(fp==NULL) /* In case the file can't be created/opened. */
{
    printf("\n\"StudentsData.txt\" file can\'t be created.\n");
    exit(0);
}
else /* When the file is created/opened. */
{
    printf("\n\"StudentsData.txt\" file successfully created.
    Your data is saved in this file.\n");
}
```

[Cont.]

```
/* Initiate writing data to "StudentsData.txt". */
/* Checking for the presence of any error before writing. */
if(ferror(fp))
{
    printf("\nCan\'t write to the file \"StudentsData.txt\".
    Some error occurred.\n");
    exit(0);
}
/* Writing if no errors. */
fprintf(fp, "Students Details (of 3 Students)\n");
fprintf(fp, "================================\n");
for (i=0;i<3;i++)
{
    fprintf(fp, "\nDetails of Student %d\n--------------------",
    i+1);
    fprintf(fp, "\nRegd. No.: %d\nName: %s\nCGPA: %.2f\n",
    studs[i].regdNo, studs[i].name, studs[i].cgpa);
}
/* Writing ends. */

/* Closing the file "StudentsData.txt". */
fclose(fp);
```

[Cont.]

```
    /* Re-opening the file "StudentsData.txt" in read mode. */
    fp = fopen("StudentsData.txt","r");

    /* Initiate reading data from "StudentsData.txt" and displaying it on the screen. */
    fprintf(stdout, "\nNow Reading data from the file
    \"StudentsData.txt\".\n\n");
    /* Checking for EOF or the presence of any error before reading. */
    if(feof(fp) || ferror(fp))
    {
       printf("\nCan\'t read from the file \"StudentsData.txt\".
       Either EOF reached or some error occurred.\n");
       exit(0);
    }
    /*Read if not EOF or no errors. */
    c = fgetc(fp);
    while (c != EOF)
    {
        printf ("%c", c);
        c = fgetc(fp);
    }
    /* Reading and displaying ends. */

    /* Closing the file "StudentsData.txt" */
    fclose(fp);
}
```

Output

```
Enter Students Information (for 3 Students)
===========================================

Student 1
---------
Regd. No.: 12345
Name: Ravi
CGPA: 9.9

Student 2
---------
Regd. No.: 15698
Name: Krish
CGPA: 8.3

Student 3
---------
Regd. No.: 15623
Name: Kartik
CGPA: 9.2

"StudentsData.txt" file successfully created. Your data is saved in this
file.
```

[Cont.]

```
Now reading data from the file "StudentsData.txt".

Students Details (of 3 Students)
================================

Details of Student 1
--------------------
Regd. No.: 12345
Name: Ravi
CGPA: 9.9

Details of Student 2
--------------------
Regd. No.: 15698
Name: Krish
CGPA: 8.3

Details of Student 3
--------------------
Regd. No.: 15623
Name: Kartik
CGPA: 9.2
```

Command Line Arguments

What are They & How are They Used?

- **What are Command Line Arguments?:**
  - Command line arguments are parameters / arguments supplied to a C program when the program is invoked from the command prompt.
  - These arguments are given after the name of the program in the command prompt.

For Example: We can write a program "CopyFromFile.c" that intends to copy the content of one file to another, where the parameters i.e., the source file name (say, "SrcFile.txt") and the target file name (say, "SrcFile.txt") are supplied to the program from the command line itself as follows:

```
C:\>CopyFromFile SrcFile TrgtFile
```

How the Command Line Arguments are Used by the Program?:

- The command line arguments are accessed and processed by the `main()` functions.
- For this, we must define the `main()` function as follows:

```
void main(int argc, char* argv[]) { /* statements */ }
Or,
int main(int argc, char* argv[]) { /* statements */ }
```

Here,

- The 1st parameter argc (called, argument count) is an integer that counts the number of arguments on the command line.
- The 2nd parameter argv (called, argument vector) is a 1-D array of pointers to strings that points to the individual arguments on the command line.

For Example,

For the command line `C:\>CopyFromFile SrcFile TrgtFile`

- argc **is** 3
- argv[0] **is (points to)** "CopyFromFile"
- argv[1] **is (points to)** "SrcFile"
- argv[2] **is (points to)** "TrgtFile".

- **Once we have the number of command line arguments (in** argc**), and the command line arguments themselves (in** argv**), we can use them in our program as we like.**
- **Note that, all the command line arguments are interpreted as strings (not any other data type).**

  So, we should not write a program that intends to process any other data types from command line (because, if we do it, they will be interpreted as strings and our purpose may not be solved)**. For example, we should not write a program to add two integers by passing the integers to the program on command line.**

Programming Examples

- **Programming Example 1:**

```
/* PR13_2.c: A program that takes some names from the command line and displays them. */

#include <stdio.h>
#include <stdlib.h>

void main(int argc, char* argv[])
{
    int i;

    printf("\n\n================================================");
    printf("\nYou have typed these names on the command line:\n");
    for (i=1; i<argc; i++)
    {
        printf("\n%s", argv[i]);
    }
    printf("\n================================================\n\n");
}
```

Command line

```
C:\>PR13_2 Odisha India
```

Output

```
================================================
You have typed these names on the command line:

Odisha
India
================================================
```

Programming Example 2:

```
/* PR13_3.c: Program that accepts a file name and a line of text as command line arguments. It first
writes the line of text to the file and then displays the contains of the file on the screen. */

# include <stdio.h>
# include <conio.h>

void main(int argc, char* argv[])
{
    char c;
    int i;
    FILE *fp;

    printf("\n\n=============================================\n");
```

[Cont.]

```
/* Creating/opening the file in write mode, whose name is given on command line (automatically
stored in argv[1]). */
fp = fopen(argv[1],"w");
if(fp==NULL)
{
    printf("\n\"%s\" file can\'t be created.\n", argv[1]);
    exit(0);
}
else
{
    printf("\n\"%s\" file successfully created.
    Your line of text is saved in this file.\n", argv[1]);
}

/* Writing the line of text entered on command line (stored in argv[2]) to the file if no errors. */
if(ferror(fp))
{
    printf("\nCan\'t write to the file \"%s\".
    Some error occurred.\n", argv[1]);
    exit(0);
}
for (i=2; i<argc; i++) /* Now Writing. */
{
    fprintf(fp, "%s ", argv[i]);
}
```

[Cont.]

```
    /* Closing the file. */
    fclose(fp);

    /* Re-opening the file in read mode. */
    fp = fopen(argv[1],"r");

    /* Reading data from the file if not EOF or no errors and displaying it on the screen. */
    fprintf(stdout, "\nNow Reading data from the file \"%s\"...\n",
    argv[1]);
    if(feof(fp) || ferror(fp))
    {
       printf("\nCan\'t read from the file \"%s\".
       Either EOF reached or some error occurred.\n", argv[1]);
       exit(0);
    }
    printf("\nYou have written the following text in the file
    \"%s\":\n", argv[1]);
    c = fgetc(fp); /* Now Reading & Displaying. */
    while (c != EOF)
    {
        printf ("%c", c);
        c = fgetc(fp);
    }
    fclose(fp); /* Closing the file. */
    printf("\n\n==========================================\n\n");
}
```

Command line

```
C:\>PR13_3 NewData.txt Hello there, how are you doing?
```

Output

```
================================================

"NewData.txt" file successfully created. Your line of text is saved
in this file.

Now Reading data from the file "NewData.txt"...

You have written the following text in the file "NewData.txt":
Hello there, how are you doing?

================================================
```

Assignments

Complete the experiments given in "Lab Manual - Section 17"**.**

End of Chapter 13