CS110 : Computer Programming Lab

Module 4 Stage 3

Practice Drill

---

1 Objective of the Stage

In this session, our goal is to make you conversant with the basics of creating, accessing, and manipulating files using the C programming language. Though the complete description of using files in C is beyond the scope of CS110, as well as that of CS101, we aim to prepare you with enough tools so that you can begin using files in your programs.

1.1 Reference Lecture Slides

1.2 Learning Goals

With this practice drill and the associated assessment problems, we expect you to internalise the rudiments of file manipulation in C. More specifically, you should be able to

- create a file;
- read and print the contents of a file;
- compute certain basic statistics of a file, like counting the number of characters;
- add some content to a file; and
- modify the contents of a file.

2 Basic file operations

Files in C are created using the `fopen` function. It takes two arguments, the first of which is the *name* of the file we want to create. The second argument specifies the way in which the file is to be opened, called the *mode*.

The `fopen` function returns something akin to a pointer to the file created. In reality, it is a pointer to a structure containing pertinent information about the created file, but we can suspend that discussion for later. Currently, we can treat whatever `fopen` returns as a pointer to the file. The data type of the returned value is `FILE *`.

2.1 Creating a file

We create a file by passing the string `"w"` as the mode to our `fopen` function; the `w` stands for *write*. An important aspect to guard against this mode is that if the name of the file we are creating is, say, *test.txt*, then `fopen` will delete any existing file of the same name. Be wary of this feature when you are manipulating a file containing important information.

To write into a file, we use the function `fprintf`. It is similar to the our familiar `printf` function except that whereas `printf` prints to the *standard output*, `fprintf` prints to a file. So, it goes without saying that we need to specify in the `fprintf` function, the file to which we want to print. We do that by specifying the file pointer as the first argument to the function. The rest of the arguments are exactly similar to what we pass to the `printf` function.

```
#include <stdio.h>

int main(void)
{
    /* declaring a file pointer */
    FILE *fp;
```

```
    /* return type - pointer to file */
    fp = fopen("test.txt", "w");

    /* what to write? */
    fprintf(fp, "hello, world!\n");

    /* close file when done */
    fclose(fp);

    return 0;
}
```

To write to a file one character at a time, we can also use the `putc` function. The following instruction writes the character `'a'` to the file pointed to by `fp`.

```
putc('a', fp);
```

2.2 Reading a file

To read a file, we open the file in *read* mode, specified in the `fopen` function as `"r"`. With this mode, the file we are trying to open must exist, otherwise an error will be generated.

To read the contents of the file we have opened, we use the counterpart of the `fprintf` function, which is the `fscanf` function. As before, we need to specify the file from which we want to read, which is specified by the file pointer as the first argument to `fscanf`. The rest of the arguments are exactly what we pass to the `scanf` function.

```
#include <stdio.h>

int main(void)
{
    /* declaring a file pointer */
    FILE *fp;

    char str[100];

    /* return type - pointer to file */
    fp = fopen("test.txt", "r");

    /* how to read? */
    fscanf(fp, "%s", str);

    printf("%s\n", str);

    /* close file when done */
    fclose(fp);

    return 0;
}
```

We can also read one character at a time using the `getc` function. It takes as argument a file pointer, reads the next character from the file, and returns it. The return type of `getc` is not `char`, but `int`. The reason will become apparent in Section 5, where we discuss the how end of file is specified.

2.3 Adding content to a file

To add content to a file that already exists, we use the append mode, denoted by `"a"`. The behaviour of this mode is, in some sense, peculiar –

it jumps to the end of the file and add contents there. So, true to its name, it appends to the contents of an existing file.

```
#include <stdio.h>

int main(void)
{
    /* declaring a file pointer */
    FILE *fp;

    /* open to append text */
    fp = fopen("test.txt", "a");

    fprintf(fp, "hello, world!\n");

    /* close file when done */
    fclose(fp);

    return 0;
}
```

2.4 Conclusion

We summarise the various modes of `fopen` discussed in the table below, and list their properties.

Mode	Description
`"r"`	It opens a file for reading. The file must exist.
`"w"`	It creates an empty file for writing. If a file with the same name already exists, its content is erased and the file is considered as a new empty file.
`"a"`	It appends to a file. Writing operations, append data at the end of the file. The file is created if it does not exist.

3 Moving about within a file

Given an open file, we can jump to different parts of the file using the `fseek` function. The first argument of the function, as was with `fprintf` and `fscanf`, is the file pointer pointing to the file within which we want to move.

The second argument is the number of bytes we want to jump from a given location. As the size of a file, measured in bytes, could be very large indeed, the second argument is a `long integer`.

The third, and the last, argument is the position from which we want to move. The table lists below the constants with which we can specify a location within a file.

Constant	Description
SEEK_SET	Beginning of file
SEEK_CUR	Current position of the file pointer
SEEK_END	End of file

To take an example, we can jump to the end of a file by using the following instruction.

```
fseek(fp, 0L, SEEK_END);
```

The way to read the instruction is as follows – within the file pointed to by `fp`, move zero bytes (`0L`) from the end (`SEEK_END`).

A function related to `fseek` is `ftell`. It takes a file pointer as an argument, and returns our current location within that file. The return value is the number of bytes to our location from the beginning of the file. To allow for large files, the return type is a `long integer`.

We can combine the `fseek` and `ftell` function to create a simple program that tells us the size of a given file.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    long nc;
```

```
        /* open file */
        fp = fopen("test.txt", "r");

        /* fseek : go to the end of the file */
        fseek(fp, 0L, SEEK END);

        /* current position */
        nc = ftell(fp);

        printf("%ld\n", nc);

        fclose(fp);

        return 0;
    }
```

A quick way to go to the beginning of a file is to use the `rewind` function. It takes a file pointer as an argument, and resets our current location to the beginning of the file (`rewind(fp)`). Observe that the same thing can be achieved by `fseek(fp, SEEK_SET, 0L)`.

4 A few more modes

The modes discussed so far, namely `r`, `w`, and `a`, has a counterpart that has a `+` at the end. These new modes allow for reading and writing into a file simultaneously.

If we want to read and write into an already existing file, we use the `"r+"` mode. In this sense, the `+` adds a new functionality to the read mode. The following code shows how to use this mode to append text to a file. So, we can use this mode in all of those places where we were using the append mode `"a"`.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;

    /* open to read and write */
    fp = fopen("test.txt", "r+");

    /* fseek : go to the end of the file */
    fseek(fp, 0L, SEEK END);

    fprintf(fp, "hello, world!\n");

    fclose(fp);

    return 0;
}
```

Similarly, the "a+" mode enhances the capabilities of the append mode and gives it the power to read a file. So, it can not only add contents at the end of a file, it can now read back the freshly written contents and everything else. The "w+" gives *read* capabilities to the "w" mode. As before, this mode erases any previous copies of the file we are opening.

Mode	Description
`"r+"`	It opens a file for reading and writing. The file must exist.
`"a+"`	It opens a file for reading and appending.
`"w+"`	It opens a file for reading and writing. It overwrites the old content.

5 End of file

The end of a file is marked by the presence of the `EOF` constant. Like the *null* character at the end of a string marking its end, the EOF constant marks the end of a file.

As the null character had to be something we were not going to use as part of a string, namely `'\0'`, the EOF constant must be something that we are not going to use in a file. It follows that it must lie outside the range of printable characters, and other special characters, and hence it must lie outside the range of character values, i.e. 0 to 255. Thus, the EOF constant is not a character, but an integer. Its value is guaranteed to be outside of the range of character values, but its exact value might vary from system to system.

This further implies that the `getc` function from Section 2.2 must be capable enough to read character values as well as the EOF value. This is the reason why the return type of the `getc` function is an `int`, and not a `char`.

6 Practice problems

In this section, we present a few problems whose solutions require one or more of the constructs discussed so far. The discussion above, combined with your programming prowess honed till now, is sufficient to crack all of the problems listed.

1. Read a file character by character and print its contents. This program will be your version of the `cat` command in Linux.
2. Write a program to make a copy of an existing file. This is similar to the `cp` command.
3. Write a program to print the number of characters, words, and lines in a given text file. This is the `wc` command in Linux.
4. Write a program to convert all uppercase letters to lowercase in a given text file.
5. Write a program to print the contents of a file backwards.
6. Write a program that compares two text files and checks whether they are identical. This will be your version of the `cmp` command.
7. Modify the above program to print the line number at which the two text files differ for the first time.
8. Write a program to capitalise the first letter of every word in a given text file.
9. Write a program to count the number of times each letter of the english alphabet occurs in a given text file.
10. Write a program to count the number of words in a given text file.