# ITC213: STRUCTURED PROGRAMMING

Bhaskar Shrestha
National College of Computer Studies
Tribhuvan University

# Lecture 13: Data Files

Readings: Chapter 12

## Data Files

- Can be created, updated, and processed by C programs
- Are used for permanent storage of large amounts of data
  - Storage of data in variables and arrays is only temporary
  - Information stored in a data file can be accessed and altered whenever necessary
- All operations between files and your programs are carried out by C I/O functions

## Program Input/Output

- A C program keeps data in RAM in the form of variables
- Data can come from some location external to the program
  - Data moved from an external location into RAM, where the program can access it, is called *input*
  - Keyboard and disk files are the most common sources of program *input*
- Data can also be sent to a location external to the program; this is called output
  - The most common destinations for output are the screen, a printer, and disk files

## I/O Devices

- Input sources and output destinations are collectively referred to as devices
- The keyboard is a device, the screen is a device, and so on
- Some devices (the keyboard) are for input only, others (the screen) are for output only, and still others (disk files) are for both input and output

- Input sources and output destinations are collectively referred to as *devices*
- The keyboard is a device, the screen is a device, and so on
- Some devices (the keyboard) are for input only, others (the screen) are for output only, and still others (disk files) are for both input and output

## Streams (1/2)

- Whatever the device, and whether it’s performing input or output, C carries out all input and output operations by means of streams
- A stream is a sequence of bytes of data
  - A sequence of bytes flowing into a program is an input stream
  - A sequence of bytes flowing out of a program is an output stream
- Advantages of Streams
  - By focusing on streams, you don’t have to worry as much about where they’re going or where they originated
  - With streams, input/output programming is device independent

## Streams (2/2)

  - Programmers don’t need to write special input/output functions for each device (keyboard, disk, and so on)
- Every C stream is connected to a file
- In this context, the term file *doesn't* refer to a disk file
  - Rather, it is an intermediate step between the stream that your program deals with and the actual physical device being used for input or output
- Because streams are largely device independent, the same function that can write to a disk file can also write to another type of device such as console
- C streams fall into two modes: text and binary

# Text Streams

- A **text stream** consists of *sequence of characters*, such as text data being sent to the screen
- Text streams are organized into lines, which can be up to 255 characters long and are terminated by an end-of-line, or newline, character
- Certain characters in a text stream are recognized as having special meaning, such as the newline character
- In a text stream, certain character translations may occur as required by the host environment
  - For example, a newline may be converted to a carriage return/linefeed pair

## Binary Streams

- A **binary stream** is a *sequence of bytes*
- A binary stream can handle any sort of data, including, but not limited to, text data
- Bytes of data in a binary stream aren’t translated or interpreted in any special way; they are read and written exactly as-is
- In text stream, end of file is determined by a special character having ASCII value 26, but in binary stream end of file is determined by the directory listing of host environment

## Predefined Streams

- ANSI C has three predefined streams
- These streams are automatically opened when a C program starts executing and are closed when the program terminates
- All standard streams are text streams
- Whenever you have used the printf or puts functions to display text on-screen, you have used the stdout stream
- Likewise, when you use gets or scanf to read keyboard input, you use the stdin stream

| **Name** | **Stream** | **Device** |
|---|---|---|
| stdin | Standard input | Keyboard |
| stdout | Standard output | Screen |
| stderr | Standard error | Screen |

## File System Basics

- The C File system is composed of several interrelated functions
- The header <stdio.h> provides the prototypes for the I/O functions and defines these three types: size_t, fpos_t and FILE
- Also defined in <stdio.h> are several macros
  - NULL, EOF, FOPEN_MAX, SEEK_SET, SEEK_CUR, SEEK_END etc

## Commonly Used C File-System Functions

| **Name** | **Function** |
|---|---|
| `fopen` | Opens a file |
| `fclose` | Closes a file |
| `putc, fputc` | Writes a character to a file |
| `getc, fgetc` | Reads a character from file |
| `fgets` | Reads a string from a file |
| `fputs` | Writes a string to a file |
| `fprintf` | Is to a file what printf is to the console |
| `fscanf` | Is to a file what scanf is to the console |
| `feof` | Returns true if end-of-file is reached |
| `ferror` | Returns true if an error has occurred |
| `fflush` | Flushes a file |

## Files

- In C, a file may be from a disk file to a terminal or printer
- You associate a stream with a specific file by performing an *open* operation
- Once a file is open, *information can be exchanged between it and your program*
  - You can read from or write to the file
- You disassociate a file from a specific stream with a *close* operation
  - When you are done working with the file, you must close the file

## Opening a File

- The process of creating a stream linked to a disk file is called opening the file
- When you open a file, it becomes available for reading, writing, or both
- The `fopen` function opens a stream for use and links a file with that stream
  - `FILE* fopen(const char *filename,const char *mode);`
  - `filename` is pointer to a string that make up a valid filename and may include a path specification
  - `mode` is pointer to string that determines how the file will be opened
- On successful, `fopen` returns the file pointer associated with that file
- If `fopen` fails to open a file, by any reason, it returns NULL

## The File Pointer

- A file pointer is a pointer to a structure of type FILE
- It points to information that defines various things about the file
  - File name, status, current position in the file etc
- The file pointer identifies a specific file and is used by the associated stream to direct operation of the I/O functions
- In order to read or write files, your program needs to use file pointers
- To obtain a file pointer by calling the fopen function

# Legal Values for Mode

| **Mode** | **Meaning** |
|---|---|
| r | Opens an existing file for reading only |
| w | Open a new file for writing only. If a file with the specified name exists, it will be destroyed and a new file is created |
| a | Opens an existing file for appending. A new file will be created if the specified file doesn’t exists. New data is appended to the end of the file. |
| r+ | Open an existing file for both reading and writing |
| w+ | Open a new file for both reading and writing. If a file with specified name exists, it will be destroyed and a new file will be created |
| a+ | Open an existing file for both reading and appending. A new file will be created if the file with the specified name does not exist. |

**Add b to specify binary mode. Default is text mode**

# fopen Examples

```
FILE *fp;
fp = fopen("c:\\test.txt", "w");
if (fp == NULL) {
   printf("Cannot open file\n");
   exit(1);
}
```

**Creates a new file C:\test.txt for writing**

```
FILE *fp;
fp = fopen("c:\\test.txt", "r+");
if (fp == NULL) {
   printf("Cannot open file\n");
   exit(1);
}
```

**Opens the file c:\test.txt, both for reading and writing**

# Closing a File

- When you’re done using the file, you must close it
- The `fclose` function closes a stream that was opened by a call to `fopen`
  - `int fclose(FILE *fp);`
  - where `fp` is the file pointer returned by the call to `fopen`
  - Returns `0` if file is successfully closed
  - Returns `EOF` is an error occurs
- When a file is closed, it flushes the buffer and does a formal OS level close
- Failure to close a stream may include lost data

```
FILE *fp;
char filename[80];
char mode[4];
char response;
do
{
   /* input filename and mode */
   printf("Enter filename: ");
   scanf(" %[^\n]", filename);
   printf("Enter mode to open: ");
   scanf("%s", mode);

   /* try to open the file */
   fp = fopen(filename, mode);
   if (fp != NULL) /* success? */
   {
      printf("Successfully opened %s in %s mode\n", filename, mode);
      fclose(fp); /* we don't do anything to the file, so close it */
   }
   else
      printf("Failed to open %s in %s mode\n", filename, mode);

   printf("Another (Y/N): ");
   scanf(" %c", &response);
} while (response == 'Y' || response == 'y');
```

## Writing and Reading to a File (1/2)

- A program that uses a disk file can write data to a file, read data from a file, or a combination of the two
- You can write data to a disk file in three ways:
- Character output
  - to save single characters or lines of characters to a file
  - Use only with text files
  - main use of character output is to save text (but not numeric) data in a form that can be read by C, as well as other programs such as word processors
- Formatted output
  - to save formatted data to a file

## Writing and Reading to a File (2/2)

  - use formatted output only with text-mode files
  - primary use of formatted output is to create files containing text and numeric data to be read by other programs such as spreadsheets or databases
  - Don't use if you want to read the data back by a C program
- Direct/Unformatted output
  - to save the contents of a section of memory directly to a disk file
  - Use for binary streams only
  - direct output is the best way to save data for later use by a C program

## Character Output: putc

- C I/O defines two equivalent functions that output a character: `putc` and `fputc`
- The `putc` Function
  - writes a single character to a specified stream
    - `int putc(int ch, FILE *fp);`
  - `ch` is the character to output (only the low order byte is used)
  - `fp` is the pointer associated with the file
  - returns the character just written if successful or **EOF** if an error occurs

```
ch = getchar();
while (ch != EOF) {
   putc(ch, fp);
   ch = getchar();
}
```

# Character Output: fputs

- Use to write a line of characters to a stream
  - Unlike `puts`, it doesn't add a newline to the end of the string
- The `fputs` function
  - `int fputs(char *str, FILE *fp);`
  - `str` is a pointer to the null-terminated string to be written
  - `fp` is the pointer to type `FILE` returned by `fopen`
  - The string pointed to by `str` is written to the file
  - Returns a nonnegative value if successful or **EOF** on error

```
gets(line);
while (*line != '\0') {
  strcat(line, "\n");
  fputs(line, fp);
  gets(line);
}
```

## Character Input: getc

- The functions `getc` and `fgetc` are identical and can be used interchangeably
- They input a single character from the specified stream
  - `int getc(FILE *fp);`
  - The argument `fp` is the pointer returned by `fopen` when the file is opened
  - The function returns the character that was input or EOF on error

```
ch = getc(fp);
while (ch != EOF) {
  putchar(ch);
  ch = getc(fp);
}
```

# Character Input: fgets

- Use `fgets` to read a line of characters from a file
  - `char *fgets(char *str, int n, FILE *fp);`
  - `str` is a pointer to a buffer in which the input is to be stored
  - `n` is the maximum number of characters to be input, and
  - `fp` is the pointer to type FILE that was returned by `fopen` when the file was opened
- `fgets` reads characters from `fp` and stores in `str`
- Characters are read until a newline is encountered or until `n-1` characters have been read, whichever occurs first
- If a newline is read, it will be part of string
- The resultant string is null terminated
- Returns `str` on successful and NULL, if an error occurs

# Searching for a text in a file

```
FILE *fp;
char filename[80];
char buffer[256], str[256];
int lineno;
...
fp = fopen(filename, "r");
...
printf("Enter string to search: ");
gets(str);

lineno = 1;
fgets(buffer, 255, fp); /* read a line */
while (!feof(fp)) {
      if (strstr(buffer, str) != NULL)
            printf("Line %d: %s\n", lineno, buffer);
      fgets(buffer, 255, fp);
      lineno++;
}
fclose(fp);
```

## Formatted Output: fprintf

- Formatted file output is done with the library function `fprintf`
  - `int fprintf(FILE *fp, char *fmt, ...);`
  - Here, `fp` is a file pointer returned by a call to `fopen`
  - `fmt` is the format string
  - `...` means, in addition to `fp` and `fmt` arguments, `fprintf` takes zero, one, or more additional arguments.
    - These arguments are the names of the variables to be output to the specified stream
- `fprintf` works just like `printf`, except that it sends its output to the stream `fp` specified in the argument list
- `fprintf` returns the number of characters actually printed. If an error occurs, a negative number is returned

```
FILE *fp;
float array[5] = {10.2, 5.0, 5.55, 6.005, 135};
int i;

fp = fopen("test.txt", "w");
if (fp == NULL) {
      printf("Unable to open output file test.txt\n");
      exit(1);
}

fprintf(fp, "Here are the %d values of the array:\n", 5);
for (i = 0; i < 5; i++)
      fprintf(fp, "array[%d] = %f\n", i, array[i]);

fclose(fp);
```

**After the program is run, the file test.txt contains**

```
Here are the 5 values of the array:
array[0] = 10.200000
array[1] = 5.000000
array[2] = 5.550000
array[3] = 6.005000
array[4] = 135.000000
```

# Formatted Input: fscanf

- For formatted file input, use the `fscanf` library function
  - `int fscanf(FILE *fp, const char *fmt, ...);`
  - `fp` is the pointer to type `FILE` returned by `fopen`
  - `fmt` is a pointer to the format string that specifies how fscanf is to read the input
  - the ellipses (`...`) indicate one or more additional arguments, the addresses of the variables where fscanf is to assign the input.
- The function `fscanf` works exactly the same as `scanf`, except that characters are taken from the specified stream rather than from keyboard
- `fscanf` returns the number of arguments that were actually assigned values. A return value of **EOF** means that a failure occurred before the first assignment was made

**This program reads a string and an integer from the keyboard and writes them to a disk file called test.txt. The program then reads the file and displays the information on the screen. After running the program, examine the test.txt file. As you will see, it contains human-readable text.**

```
FILE *fp;
char s[80];
int t;
fp = fopen("test.txt", "w");
if (fp == NULL) {
  printf("Cannot open file\n");
  exit(1);
}
printf("Enter a string and number: ");
scanf("%s%d", s, &t);
fprintf(fp, "%s %d", s, t); /* write to file */
fclose(fp);
fp = fopen("test.txt", "r");
if (fp == NULL) {
  printf("Cannot open file\n");
  exit(1);
}
fscanf(fp, "%s%d", s, &t); /* read from file */
printf("%s %d", s, t);
```

## Direct (Unformatted) I/O

- Use direct file I/O,  when you want save data to be read later by the same or a different C program
- Direct I/O is used only with binary-mode files
- With direct output, blocks of data are written from memory to disk
- Direct input reverses the process:
  - A block of data is read from a disk file into memory
- The direct I/O functions are `fread` and `fwrite`

## The fwrite function

- The `fwrite` library function writes a block of data from memory to a file
  - `int fwrite(void *buf, int size, int count, FILE *fp);`
  - `buf` is a pointer to the first byte of memory holding the data to be written to the file
  - `size` specifies the size, in bytes, of the individual data items, and `count` specifies the number of items to be written
  - `fp` is the pointer to type `FILE`, returned by `fopen` when the file was opened
- The `fwrite` function returns the number of items written on success; if the value returned is less than count, it means that an error has occurred

## fwrite examples

```
double x = 12e-6;
/* write a double value x */
fwrite(&x, sizeof(double), 1, fp);

int arr[10];
...
/* write an array arr*/
fwrite(arr, sizeof(int), 10, fp);

struct address addr[50], myaddr;
...
/* write a structure variable myaddr*/
fwrite(&myaddr, sizeof(struct address), 1, fp);
/* write the entire structure array addr */
fwrite(addr, sizeof(struct address), 50, fp);
```

## The fread function

- The `fread` library function reads a block of data from a file into memory
  - `int fread(void *buf, int size, int count, FILE *fp);`
  - `buf` is a pointer to the first byte of memory that receives the data read from the file
  - `size` specifies the size, in bytes, of the individual data items being read, and count specifies the number of items to read
  - `fp` is the pointer to type `FILE` that was returned by `fopen` when the file was opened
- The `fread` function returns the number of items read; this can be less than count if end-of-file was reached or an error occurred

## fread examples

```
double x;
/* read a double value x */
fread(&x, sizeof(double), 1, fp);

int arr[10];
...
/* read an array arr*/
fread(arr, sizeof(int), 10, fp);

struct address addr[50], myaddr;
...
/* read a structure variable myaddr*/
fread(&myaddr, sizeof(struct address), 1, fp);
/* read entire structure array addr */
fread(addr, sizeof(struct address), 50, fp);
```

## File Buffer

- When a stream linked to a disk file is created, a buffer is automatically created and associated with the stream
- A buffer is a block of memory used for temporary storage of data being written to and read from the file
- Buffers are needed because disk drives are block-oriented devices, which means that they operate most efficiently when data is read and written in blocks of a certain size
- The size of the ideal block differs, depending on the specific hardware in use

## Use of Buffer

- Buffer serves as an interface between the stream and the disk hardware
- As a program writes data to the stream, the data is saved in the buffer until the buffer is full, and then the entire contents of the buffer are written, as a block, to the disk
- An analogous process occurs when reading data from a disk file
- During program execution, data that a program wrote to the disk might still be in the buffer, not on the disk
  - If your program hangs up, if there's a power failure, or if some other problem occurs, the data that's still in the buffer might be lost, and you won't know what's contained in the disk file

## Flushing Buffer (1/2)

- For a output stream, flushing buffer  means writing the buffered data to the file and clearing it contents
- For an input stream, flushing buffer means clearing its buffer contents
- To flush a buffer contents use `fflush`
  - `int fflush(FILE* fp);`
  - `fp` is a file pointer to stream, whose buffer is to be flushed
  - Returns `0` if successful, `EOF` otherwise
  - If `fp` is **NULL**, all files opened for output are flushed

## Flushing Buffer (2/2)

- The buffer is automatically flushed, when a stream is closed with a call to `fclose`
- All opened files are automatically closed when the program terminates normally, hence all opened streams are flushed
- Use `flushall` to flush the buffers of all open streams
  - `int flushall(void);`
  - returns the number of open streams

## Sequential Versus Random File Access (1/2)

- Every open file has a file position indicator associated with it
- Position indicator specifies where read and write operations take place in the file
- The position is always given in terms of bytes from the beginning of the file
  - When a new file is opened, the position indicator is always at the beginning of the file, position 0
  - When an existing file is opened, the position indicator is at the end of the file if the file was opened in append mode, or at the beginning if the file was opened in any other mode

## Sequential Versus Random File Access (2/2)

- Input/output functions make use of the position indicator
- Writing and reading operations occur at the location of the position indicator and update the position indicator as well
- For example, if you open a file for reading, and 10 bytes are read, you read the first 10 bytes in the file (the bytes at positions 0 through 9). After the read operation, the position indicator is at position 10, and the next read operation begins there

## The ftell and fseek function

- Use `ftell` and `fseek` to determine and change the value of the file position indicator
- By controlling the position indicator, you can perform random file access
  - Here, random means that you can read data from, or write data to, any position in a file without reading or writing all the preceding data
- The `ftell` function
  - `long ftell(FILE *fp);`
  - Returns the location of the current position of the file associated with `fp`.
  - If a failure occurs, returns **-1**

## The fseek function

- The `fseek` function sets the file position indicator
- `int fseek(FILE *fp, long numbytes, int origin);`
  - `fp` is a file pointer returned by `fopen`
  - `numbytes` is the number of bytes from origin, which will become the end current position
  - `origin` can be one of the macros shown in the table

| **Origin** | **Macro Name** |
|---|---|
| Beginning of file | SEEK_SET |
| Current Position | SEEK_CUR |
| End of file | SEEK_END |

# Examples of ftell and fseek

```
long pos;
/* seek 0 bytes from beginning of file */
fseek(fp, 0, SEEK_SET);
pos = ftell(fp);
printf("%ld\n", pos); /* prints 0 */

/* seek 0 bytes from end of file */
fseek(fp, 0, SEEK_END);
pos = ftell(fp);
printf("%ld\n", pos); /* prints file size in bytes */

/* seek –pos bytes from current position */
fseek(fp, -pos, SEEK_CUR);
pos = ftell(fp);
printf("%ld\n", pos); /* prints 0 why? */
```

# Other file related functions

| **Name** | **Function** |
|---|---|
| `clearerr` | Resets the error flag of the associated stream |
| `fgetpos` | Gets the current value of file position indicator |
| `freopen` | Opens an existing opened file in another mode |
| `fsetpos` | Moves the file position indicator |
| `remove` | Deletes a file |
| `rename` | Changes the name of an existing file |
| `rewind` | Resets the file position indicator to beginning of file |
| `setbuf` | Sets the buffer to use or turns off buffering |
| `tmpfile` | Opens a temporary file for reading/writing |
| `tmpnam` | Generates a unique filename |
| `ungetc` | Puts a character in the buffer, such that it can be obtained from next read operation |