



File Handling

Storage seen so far

- All variables stored in memory
- Problem: the contents of memory are wiped out when the computer is powered off
- Example: Consider keeping students' records
 - 100 students records are added in array of structures
 - Machine is then powered off after sometime
 - When the machine is powered on, the 100 records entered earlier are all gone!
 - Have to enter again if they are needed

Solution: Files

- A named collection of data, stored in secondary storage like disk, CD-ROM, USB drives etc.
- Persistent storage, not lost when machine is powered off
- Save data in memory to files if needed (file write)
- Read data from file later whenever needed (file read)

Organization of a file

- Stored as sequence of bytes, logically contiguous
 - May not be physically contiguous on disk, but you do not need to worry about that
- Two kinds of files:
 - **Text** : contains ASCII codes only
 - **Binary** : can contain non-ASCII characters
 - Example: Image, audio, video, executable, etc.
- We will do text files only

Basic operations on a file

- Open
- Read
- Write
- Close
- Mainly we want to do read or write, but a file has to be opened before read/write, and should be closed after all read/write is over

Opening a File: `fopen()`

- `FILE *` is a datatype used to represent a pointer to a file
- `fopen` takes two parameters, the name of the file to open and the `mode` in which it is to be opened
- It returns the pointer to the file if the file is `opened` successfully, or `NULL` to indicate that it is unable to open the file

Example: opening file.dat for write

```
FILE *fptr;  
char filename[ ]= "file2.dat";  
fptr = fopen (filename,"w");  
if (fptr == NULL) {  
    printf ("ERROR IN FILE CREATION");  
    /* DO SOMETHING */  
}
```

Modes for opening files

- The second argument of `fopen` is the `mode` in which we open the file.

Modes for opening files

- The second argument of `fopen` is the `mode` in which we open the file.
 - `"r"` : opens a file for reading (can only read)
 - Error if the file does not already exist
 - `"r+"` : allows write also

Modes for opening files

- The second argument of `fopen` is the `mode` in which we open the file.
 - `"r"` : opens a file for reading (can only read)
 - Error if the file does not already exist
 - `"r+"` : allows write also
 - `"w"` : creates a file for writing (can only write)
 - Will create the file if it does not exist
 - **Caution:** writes over all previous contents if the file already exists
 - `"w+"` : allows read also

Modes for opening files

- The second argument of `fopen` is the `mode` in which we open the file.
 - `"r"` : opens a file for reading (can only read)
 - Error if the file does not already exist
 - `"r+"` : allows write also
 - `"w"` : creates a file for writing (can only write)
 - Will create the file if it does not exist
 - **Caution:** writes over all previous contents if the file already exists
 - `"w+"` : allows read also
 - `"a"` : opens a file for appending (write at the end of the file)
 - `"a+"` : allows read also

The `exit()` function

- Sometimes error checking means we want an **emergency exit** from a program
- Can be done by the `exit()` function
- The `exit()` function, called from anywhere in your C program, will terminate the program at once

Usage of exit()

```
FILE *fptr;  
char filename[]= "file2.dat";  
fptr = fopen (filename,"w");  
if (fptr == NULL) {  
    printf ("ERROR IN FILE CREATION");  
    /* Do something */  
    exit(-1);  
}  
.....rest of the program.....
```

Writing to a file: `fprintf()`

- `fprintf()` works **exactly like `printf()`**, except that its first argument is a file pointer. The remaining two arguments are the same as `printf`
- The behaviour is **exactly the same**, except that the writing is done on the file instead of the display

```
FILE *fptr;  
fptr = fopen ("file.dat", "w");  
fprintf (fptr, "Hello World!\n");  
fprintf (fptr, "%d %d", a, b);
```

Reading from a file: `fscanf()`

- `fscanf()` works like `scanf()`, except that its first argument is a file pointer. The remaining two arguments are the same as `scanf`
- The behaviour is **exactly the same**, except
 - The reading is done from the file instead of from the keyboard (think as if you typed the same thing in the file as you would in the keyboard for a `scanf` with the same arguments)
- Returns the number of variables successfully read, the special value **EOF** if no variable can be read because end of file is reached (nothing left to read)

Reading from a file: fscanf()

```
FILE *fptr;
fptr = fopen ("input.dat", "r");
/* Check it's open */
if (fptr == NULL)
{
    printf("Error in opening file \n");
    exit(-1);
}
fscanf (fptr, "%d %d",&x, &y);
```

EOF checking in a loop

```
char ch;
while (fscanf(fptr, "%c",
&ch) != EOF)
{
    /* not end of file; read */
}
}
```

Reading lines from a file: `fgets()`

- Takes three parameters
 - a character array `str`, maximum number of characters to read `size`, and a file pointer `fp`
- Reads from the file `fp` into the array `str` until **any one** of these happens
 - No. of characters read = `size` - 1
 - `\n` is read (the char `\n` is added to `str`)
 - EOF is reached or an error occurs
- `'\0'` added at end of `str` if no error
- Returns NULL on error or EOF, otherwise returns pointer to `str`

Reading lines from a file: `fgets()`

```
FILE *fptr;  
char line[1000];  
/* Open file and check it is open */  
while (fgets(line,1000,fptr) != NULL)  
{  
    printf ("Read line %s\n",line);  
}
```

Writing lines to a file: `fputs()`

- Takes two parameters
 - A string `str` (null terminated) and a file pointer `fp`
- Writes the string pointed to by `str` into the file
- Returns non-negative integer on success, EOF on error

Reading/Writing a character: fgetc(), fputc()

- Equivalent of `getchar()`, `putchar()` for reading/writing char from/to keyboard
- Exactly same, except that the first parameter is a file pointer
- Equivalent to reading/writing a byte (the char)

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

```
int fputc(int c, FILE *fp);
```

- Example:

```
char c;
```

```
c = fgetc(fp1); fputc(c, fp2);
```

Formatted and Un-formatted I/O

■ Formatted I/O

- Using fprintf/fscanf
- Can specify format strings to directly read as integers, float etc.

■ Unformatted I/O

- Using fgets/fputs/fgetc/fputc
- No format string to read different data types
- Need to read as characters and convert explicitly

Closing a file

- Should close a file when no more read/write to a file is needed in the rest of the program
- File is closed using `fclose()` and the file pointer

```
FILE *fptr;  
char filename[] = "myfile.dat";  
fptr = fopen (filename, "w");  
fprintf (fptr, "Hello World of filing!\n");  
.... Any more read/write to myfile.dat....  
fclose (fptr);
```



Command Line Arguments

What are they?

- A program can be executed by directly typing a command with parameters at the prompt

```
$ cc -o test test.c
```

```
$ ./a.out in.dat out.dat
```

```
$ prog_name param_1 param_2 param_3
```

```
..
```

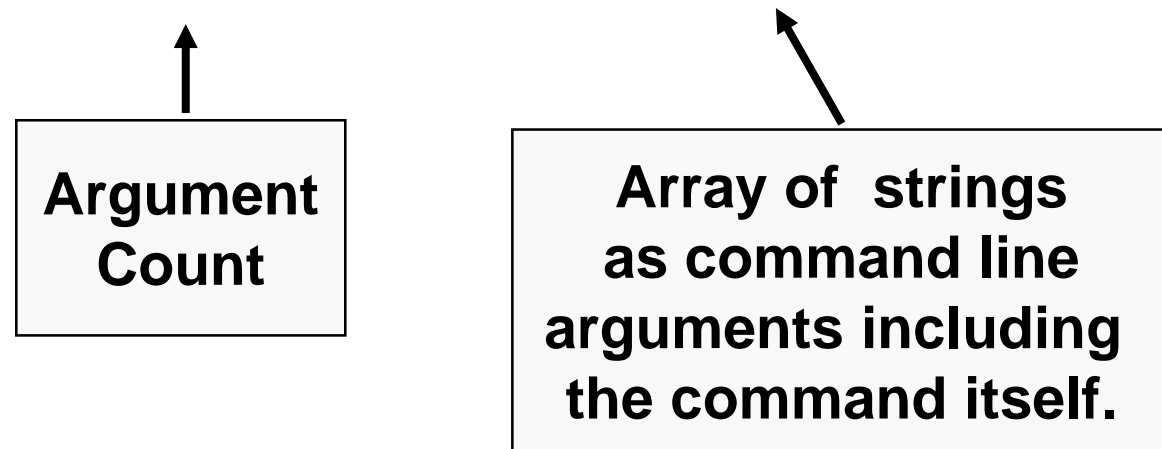
- The individual items specified are separated from one another by spaces
 - First item is the program name

What do they mean?

- Recall that `main()` is also a function
- It can also take parameters, just like other C function
- The items in the command line are passed as parameters to `main`
- Parameters `argc` and `argv` in `main` keeps track of the items specified in the command line

How to access them?

```
int main (int argc, char *argv[]);
```



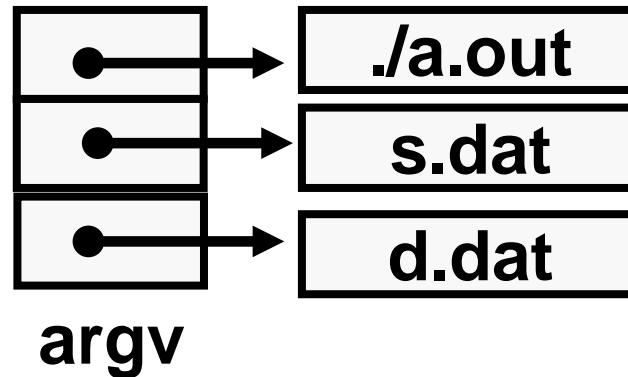
The parameters are filled up with the command line arguments typed when the program is run

They can now be accessed inside `main` just like any other variable

Example: Contd.

```
$ ./a.out s.dat d.dat
```

argc=3



`argv[0] = “./a.out”`

`argv[1] = “s.dat”`

`argv[2] = “d.dat”`

Contd.

- Still there is a problem
 - All the arguments are passed as strings in `argv[]`
 - But the intention may have been to pass an `int/float` etc.
- Solution: Use `sscanf()`
 - Exactly same as `scanf`, just reads from a string (`char *`) instead of from the keyboard
 - The first parameter is the string pointer, the next two parameters are **exactly the same as `scanf`**

Example

- Write a program that takes as command line arguments 2 integers, and prints their sum

```
int main(int argc, char *argv[ ]  
{  
    int i, n1, n2;  
    printf("No. of arg is %d\n", argc);  
    for (i=0; i<argc; ++i)  
        printf("%s\n", argv[i]);  
    sscanf(argv[1], "%d", &n1);  
    sscanf(argv[2], "%d", &n2);  
    printf("Sum is %d\n", n1 + n2);  
    return 0;  
}
```

```
$ ./a.out 32 54  
No. of arg is 3  
./a.out  
32  
54  
Sum is 86
```