File Handling

What is a file?

- A named collection of data, stored in *secondary storage* (typically).
- Typical operations on files:
  - Open
  - Read
  - Write
  - Close
- How is a file stored?
  - Stored as sequence of bytes, logically contiguous (may not be physically contiguous on disk).

- The last byte of a file contains the end-of-file character (`EOF`), with ASCII code 1A (hex).
- While reading a text file, the `EOF` character can be checked to know the end.

- Two kinds of files:

a) Text :: contains ASCII codes only

b) Binary :: can contain non-ASCII characters

- Image, audio, video, executable, etc.
- To check the end of file here, the *file size* value (also stored on disk) needs to be checked.

File handling in C

- In C we use **FILE*** to represent a pointer to a file.
- **fopen** is used to open a file. It returns the special value **NULL** to indicate that it is unable to open the file.

```
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.
- There are three modes.

"r" opens a file for reading.

"w" creates a file for writing, and writes over all previous contents (deletes the data so be careful!).

"a" opens a file for appending – writing at the end of the file.

- We can add a “b” character in addition to indicate that the file is a ***binary*** file.
  - “rb”, “wb” or “ab”

```
fptr = fopen ("xyz.jpg", "rb");
```

The `exit()` function

- Sometimes error checking means we want an "*emergency exit*" from a program.
- In `main()` we can use `return` to stop.
- In functions we can use `exit()` to do this.
- Exit is part of the `stdlib.h` library.

```
exit(-1);
```

in a function is exactly the same as

```
return -1;
```

in the main routine

Usage of `exit( )`

```
FILE *fptr;
char filename[]= "file2.dat";
fptr = fopen (filename,"w");
if (fptr == NULL) {
  printf ("ERROR IN FILE CREATION");
  exit(-1);
}
………
```

Writing to a file using fprintf( )

- fprintf() works just like printf() and sprintf() except that its first argument is a file pointer.

```
FILE *fptr;
fptr = fopen ("file.dat","w");
if (fptr == NULL)
   {
     printf("Error in opening file \n");
     exit (-1);
   }
fprintf (fptr, "Hello World!\n");
fprintf (fptr, "%d %d", a, b);
```

Reading Data Using `fscanf( )`

- We also read data from a file using `fscanf()` .

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

Reading lines from a file using `fgets( )`

We can read a string from a file using `fgets()`.

```
FILE *fptr;
char line [1000];
/*** Open the file ***/
while (fgets(line,1000,fptr) != NULL)
{
   printf ("Reading line: %s\n", line);
}
```

`fgets()` takes 3 arguments – a string, maximum number of characters to read, and a file pointer.

It returns `NULL` if there is an error (such as `EOF`).

Closing a file

- We can close a file simply using `fclose()` and the file pointer.

```
FILE *fptr;
char filename[]= "myfile.dat";
fptr = fopen (filename,"w");
if (fptr == NULL) {
    printf ("Cannot open file to write!\n");
    exit(-1);
}
fprintf (fptr,"Hello World of filing!\n");
fclose (fptr);
```

Three special streams

Three special streams

- Three special file streams are defined in the `<stdio.h>` header:
  a) stdin reads input from the keyboard
  b) stdout send output to the screen
  c) stderr prints errors to an error device (usually also the screen)

- What might this do?

```
fprintf (stdout, "Hello World!\n");
```

An example program

```
#include <stdio.h>
main()
{
   int i;

   fprintf (stdout, "Give value of i \n");
   fscanf (stdin, "%d", &i);
   fprintf (stdout, "Value of i=%d \n", i);
   fprintf (stderr, "No error: But an example to
     show error message.\n");
 }
```

```
Give value of i
15
Value of i=15
No error: But an example to show error message.
```

Reading and Writing a character

- Reading or writing a character is equivalent to reading or writing a byte.

```
int getchar();
int putchar(int c);
```

stdin, stdout

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

file

- Example:

```
char c;
c = getchar();
putchar(c);
```

Example: use of `getchar()` and `putchar()`

```
#include <stdio.h>
main()
{
  int c;

  printf("Type text and press return to
    see it again \n");
  printf("For exiting press <CTRL D> \n");
  while((c = getchar()) != EOF)
     putchar(c);
}
```

Input File & Output File redirection

- One may redirect the standard input and standard output to other files (other than `stdin` and `stdout`).
- Usage: Suppose the executable file is `a.out`:

```
$ ./a.out < {input file name}  >
{output file name}
```

`scanf()` will read data inputs from the file {input file name}, and `printf()` will output results on the file {output file name}.

A Variation

```
$ ./a.out < {input file name}  >>
{append file name}
```

`scanf()` will read data inputs from the file `{input file name}`, and `printf()` will *append* results at the end of the file `{append file name}`.

Command Line Arguments

What are they?

- A program can be executed by directly typing a command at the operating system prompt.

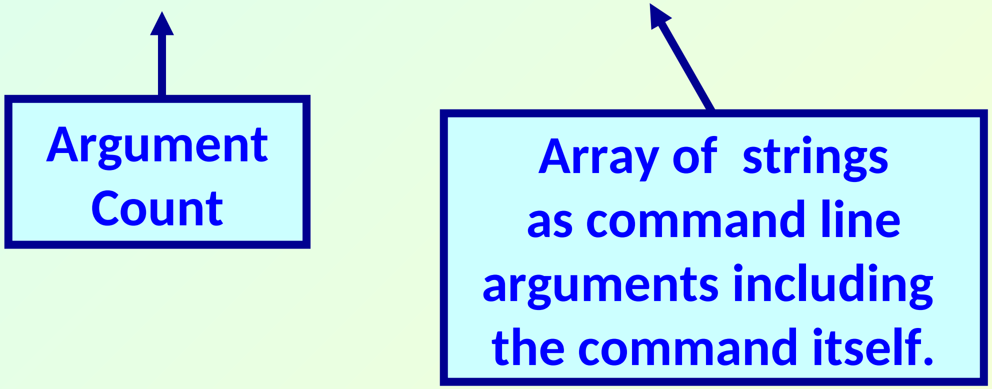
```
$ gcc –o test test.c
$ ./a.out in.dat out.dat
$ prog_name param_1 param_2 param_3 ..
```

  - The individual items specified are separated from one another by spaces.
    - First item is the program name.
  - Variables *argc* and *argv* keep track of the items specified in the command line.

How to access them?

- Command line arguments may be passed by specifying them under `main()`.

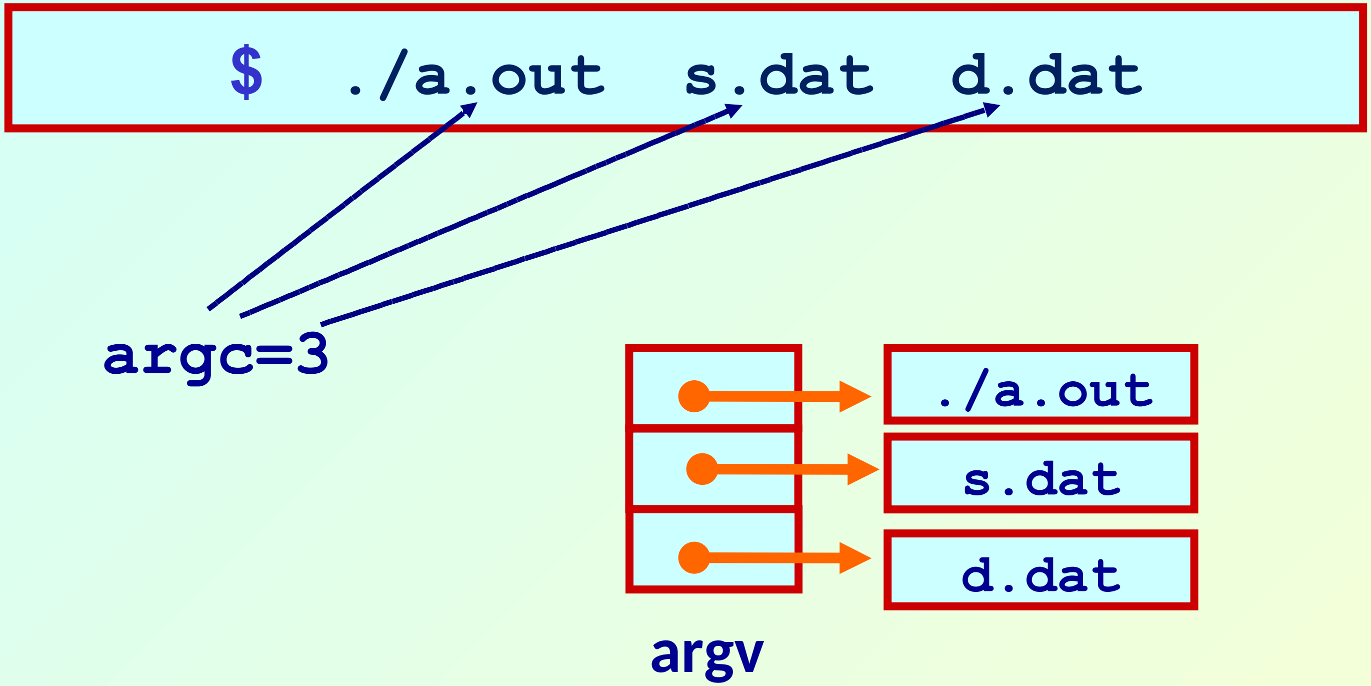
```
int main (int argc, char *argv[]);
```



Argument Count

Array of strings as command line arguments including the command itself.

Example: Contd.



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

argc=3

./a.out

s.dat

d.dat

argv

```
argv[0] = "./a.out"  argv[1] = "s.dat"
argv[2] = "d.dat"
```

Example: reading command line arguments

```
#include <stdio.h>
#include <string.h>

int main(int argc,char *argv[])
{
  FILE *ifp, *ofp;
  int i, c;
  char src_file[100],dst_file[100];

  if(argc!=3) {
    printf ("Usage: ./a.out <src_file> <dst_file> \n");
    exit(0);
  }
  else {
    strcpy (src_file, argv[1]);
    strcpy (dst_file, argv[2]);
  }
  if ((ifp = fopen(src_file,"r")) == NULL)   {
     printf ("File does not exist.\n");
     exit(0);
  }

  if ((ofp = fopen(dst_file,"w")) == NULL)   {
     printf ("File not created.\n");
     exit(0);
  }

  while ((c = fgetc(ifp)) != EOF) {
     fputc (c, ofp);
  }

  fclose(ifp);
  fclose(ofp);
}
```

Example: with command-line arguments

- Write a program which will take the number of data items, followed by the actual data items on the command line, and print the average.

```
$ ./a.out 6 10 17 35 12 28 33
```

No. of data items

argv[2] = “10”

argv[3] = “17”, and so on

Getting numbers from strings

- Once we have got a string with a number in it (either from a file or from the user typing) we can use `atoi` or `atof` to convert it to a number.
- The functions are part of `stdlib.h`

```
char numberstring[]= "3.14";
int i;
double pi;
pi = atof (numberstring);
i = atoi ("12");
```

Both of these functions return 0 if they have a problem.

- Alternatively, we can use `sscanf()` .
- For example, if

  `argv[2]="10"` and `argv[3]="17"`,

  then we can read their values into integer variables as:

```
sscanf (argv[2], "%d", &n1);
sscanf (argv[3], "%d", &n2);
```