

File Handling I

- a file pointer:

```
FILE *in = NULL, *out = NULL;
```

- opening a file:

```
in = fopen("in.txt", "r");  
out = fopen("out.txt", "w");  
out = fopen(test_out, "a");
```

"r": read mode, "w": (over-)write mode, "a": append mode

- closing a file:

```
fclose(in);  
fclose(out);
```

File Handling II

- reading from an open file (i.e. a valid FILE pointer):

```
float tmp;
```

```
fscanf(in, "%f", &tmp);
```

- its a very common mistake to use tmp instead of &tmp which would cause:
Segmentation fault
- writing to an open file (i.e. a valid FILE pointer):

```
fprintf(out, "%f", tmp);
```

File Handling III

- a small example:

```
#define SMALL_LEN 5

int
main (int argc, char **argv)
{
    int ii;
    FILE *in = NULL, *out = NULL;
    float tmp;
    double dval_arr[SMALL_LEN];

    in = fopen("in.txt", "r");
    for (ii = 0; ii < SMALL_LEN; ++ii) {
        fscanf(in, "%f", &tmp);
        dval_arr[ii] = tmp;
    }
    fclose(in);

    out = fopen("out.txt", "w");
    for (ii = 0; ii < SMALL_LEN; ++ii)
        fprintf(out, "%g\n", dval_arr[ii]);
    fclose(out);
    return 0;
}
```

File Handling IV

- some special FILE pointers:

```
stdin
stdout
stderr
```

- these are always open: `stdin`, in "r" mode and `stdout`, `stderr` in "w" mode
- is there a difference between the two statements?

```
float tmp;
```

```
scanf("%f", &tmp);
fscanf(stdin, "%f", &tmp);
```

- is there a difference between the two statements?

```
printf("%f", tmp);
fprintf(stdout, "%f", tmp);
```

- nope! `scanf` is attached to `stdin` and `printf` is attached to `stdout`

File Handling V

- the stream `stdout`, `stderr` and for that matter any output file stream i.e. a `FILE *` opened in “w” mode are always “buffered”
- “buffered” stream means that data coming to that stream is stored in a buffer for printing and its not printed until the buffer is full
- you need to use the `fflush()` function if you want to see the effect of “outputting” immediately

Aside I

- some interesting tricks:

- printing to a string:

```
int ival = 1;
double dval = 1.11;
char *str1 = "This is ", str2[MAX_WORD_LEN];
```

```
sprintf(str2, "%s%d", str1, ival);
printf("%s\n", str2);
sprintf(str2, "%s%g", str1, dval);
printf("%s\n", str2);
```

- the above should give:

```
This is 1
This is 1.11
```

Aside II

- some interesting tricks:
 - scanning from a string:

```
int jj;
float ff;
char cc[SMALL_LEN];
char str3[ ] = "12, 3.4, gopi";
char str4[ ] = "gopi :: 12, 3.4";
char str5[ ] = "gopi = 12";

sscanf(str3, "%d, %f, %s", &jj, &ff, &cc);
printf("jj = %d, ff = %f, cc = %s\n", jj, ff, cc);
sscanf(str4, "%s :: %d, %f", &cc, &jj, &ff);
printf("jj = %d, ff = %f, cc = %s\n", jj, ff, cc);
sscanf(str5, "%[a-zA-Z0-9] = %d", &cc, &jj);
printf("jj = %d, cc = %s\n", jj, cc);
```

- the above would give:

```
jj = 12, ff = 3.400000, cc = Harry
jj = 12, ff = 3.400000, cc = Harry
jj = 12, cc = Harry
```

Aside III

- some useful string to int, double functions:

```
int
```

```
atoi(const char *nptr);
```

```
double
```

```
atof(const char *nptr);
```

```
long
```

```
strtol(const char * restrict nptr, char ** restrict endptr, int base);
```

Aside IV

- some other useful function you would find handy while:
 - reading command line arguments: `getopt()`
 - processing a “configuration file”: `fgets()`, `strtok_r()`
- usage of `getopt()` and `strtok_r()` is a bit tricky and so skipping the details here, consult google or me if you are really interested
- if you are doing things from within R then you wouldn't need these functions *ever*, may be!

Code Files

prog7.c