

Introduction to C, C++, and Unix/Linux

CS 60

File I/O

Today

→ File I/O

File I/O

- We've already seen I/O functions that write to or read from the standard input/output:
 - **printf, scanf, putchar, getchar, gets**
- These have related functions that write to / read from files:
 - **fprintf, fscanf, putc, getc, fgets**
- And some that write to / read from strings
 - **sprintf, sscanf**

sprintf and **sscanf**

- “String versions” of **printf** and **scanf**
- Just like **printf** and **scanf**, but there’s an additional parameter: the target string (character array)

```
sprintf(char *str, const char *format, ...)  
sscanf(char *str, const char *format, ...)
```

```
char str1[ ] = "1 2 3 go";
char str2[100], s[100];
int a, b, c;

sscanf(str1, "%d%d%d%s", &a, &b, &c, s);
sprintf(str2, "%s %s %d %d %d\n",
        s, s, a, b, c);

printf("%s", str2); → "go go 1 2 3"
```

fprintf and **fscanf**

- “File versions” of **printf** and **scanf**
- Just like **printf** and **scanf**, but there’s an additional parameter: a file pointer

```
fprintf(FILE *fp, const char *format, ...)  
fscanf(FILE *fp, const char *format, ...)
```


fp is a “file pointer”

It points to a **struct** that describes the file

The file must be (1) opened and (2) ready for writing/reading

Opening a file – `fopen`

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



or

```
if ((fp=fopen("data.txt", "r"))== NULL) {...}
```

From here on out, all file operations are done on the file pointer `fp`

`fopen` mode string

- The “mode” string can be one of several options:
 - “r” open text file for reading
 - “w” create text file for writing
 - “a” append to text file
 - “r+” open text file for reading/writing
 - “w+” create text file for reading/writing
 - “a+” append or create text file for appending
- To specify a binary file, append “b” to these
 - “rb”, “wb”, “ab”, “r+b”, “w+b”, “a+b”

Closing a file – `fclose`

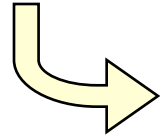
- An open file should be closed before exiting
 - `fclose(fp)`
 - Ensures the associated buffer is properly read/written
 - ◆ This can be done with `fflush(fp)`
 - Allows other programs to safely access the file
 - On a proper program exit, all the files are closed – but if there's an improper/unexpected exit all bets are off (data may be corrupted)

Standard input/output/error

- C defines three “files” always open and ready for your use (declared in `<stdio.h>`)
 - **stdin** – standard input
 - **stdout** – standard output
 - **stderr** – standard error (for printing diagnostic and error messages)
- These are for command-line input and output

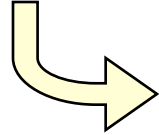
Equivalent calls:

```
printf("Hi world")
```



```
fprintf(stdout, "Hi world")
```

```
scanf("%d", &x)
```



```
fscanf(stdin, "%d", &x)
```

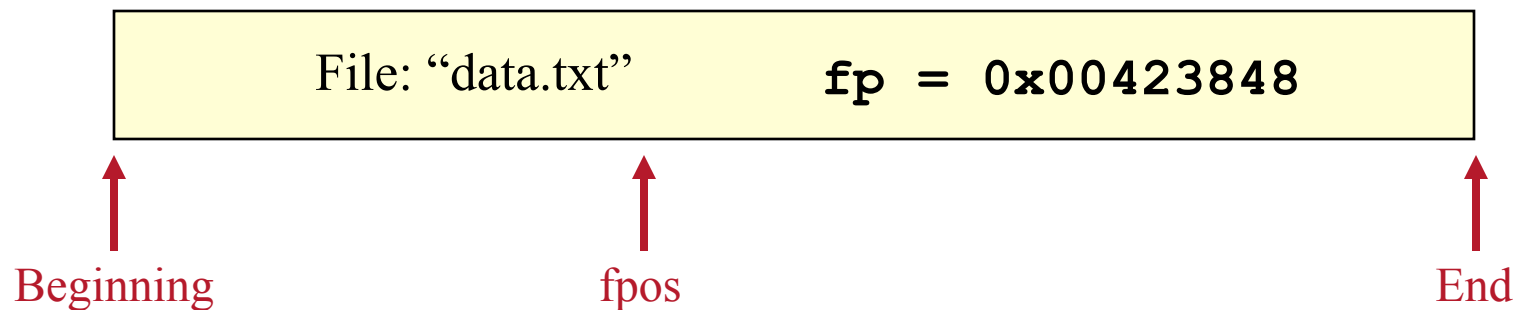
```
#include <stdio.h>
```

Example: Reading from a file

```
int main (void) {  
    FILE *fp;  
    char c;  
  
    if ((fp=fopen("data.txt", "r")) == NULL) {  
        fprintf(stderr, "Unable to open file\n");  
        exit(1);  
    }  
    do {  
        c = (char)getc(fp);  
        fprintf(stdout, "%c", c);  
    } while (c != EOF);  
  
    return( fclose(fp) );  
}
```

File model

- An open file is considered as a data stream, with a beginning, an end, and a current location
 - Called the file position indicator (fpos)
 - fpos indicates where you are currently reading/writing



I/O functions

Arsenal of file I/O related calls:

<code>fopen()</code>	-	open file
<code>fclose()</code>	-	close file
<code>fputc()</code>	-	write char to file
<code>fgetc()</code>	-	grab char from file
<code>fread()</code>	-	reads data from file
<code>fwrite()</code>	-	writes data to file
<code>fseek()</code>	-	seek to specified byte in file
<code>fprintf()</code>	-	print to file stream
<code>fscanf()</code>	-	scanf() for a file stream
<code>feof()</code>	-	check if end-of-file reached
<code>ferror()</code>	-	if error has occurred
<code>remove()</code>	-	nuke a file

File I/O using Unix system calls

Arsenal of file I/O related system calls:

```
int open(char *name, int flags, int perms);  
int close(int fd);  
int creat(char *name, int perms);  
int unlink(char *pathname);  
int read(int fd, char *buf, int n);  
int write(int fd, char *buf, int n);  
long lseek(int fd, long offset, int origin);
```

For more fine-grained control...

But for many applications, the stdlib functions (fopen, fread, etc.) are sufficient – and they're more portable!