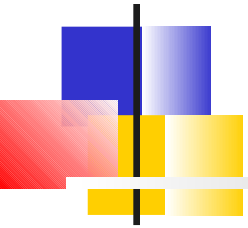
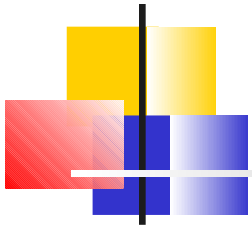


File Input and Output

Chapter 22



Objectives

You will be able to

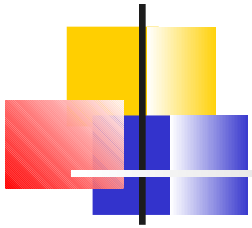
- Write C programs that read text data from disk files.
- Write C programs that write text data to disk files.

We consider only *text* I/O.

Binary I/O is a more advanced topic.

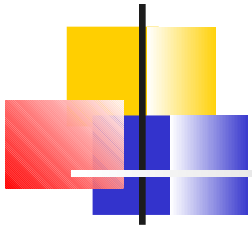
Also *random access* I/O

Not covered in this course.



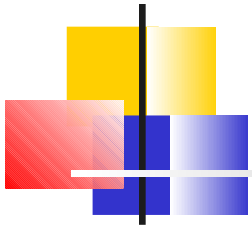
File I/O

- Text File Input and Output is similar to Keyboard Input and Screen Output.
  - Have to identify the file.
- Before we can read or write a file, we have to *open* the file.
  - fopen() Standard IO Library Function
    - Call to fopen() identifies a file.
    - Specifies what we want to do with the file.
    - Creates a data structure used by I/O library functions.
    - Returns a pointer to that data structure.



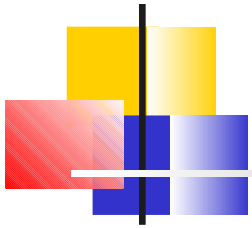
The FILE Struct

- fopen creates a FILE struct and returns a pointer to it.
  - Note all caps.



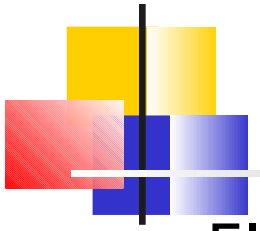
The FILE Struct

- FILE is a struct defined in stdio.h
  - Used by the standard IO library functions on file read and write operations.
  - Each file read or write operation passes a pointer to a FILE struct to a library function.
  - Details of the struct definition are of no concern to us.
    - An *opaque* object.



The FILE Struct

- A file can be open for reading by any number of different programs at the same time.
  - Each has a separate FILE struct that holds information such as where the next read should occur.
- A FILE corresponds to a single instance of *reading* or *writing* a file.
  - *Stream* is more descriptive.
  - FILE is standard terminology.



The fopen() function

FILE* fopen (char* **filename**, char* **mode**)

filename

Text string just as we would use at the command line

Can specify just name of a file in the current working directory

Example: "test.txt"

Can specify full path.

Example: "/home/rtindell/Programming/Ch_15/test.txt"

mode

What we want to do

"r" to Read

"w" to Write

"a" to Append

...

The fopen() function

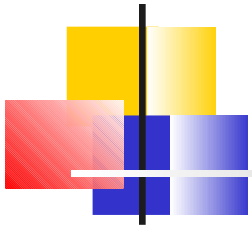
Example:

```
FILE* pFile;

pFile = fopen("test.c", "r");
```

File Name (Current Working Directory)

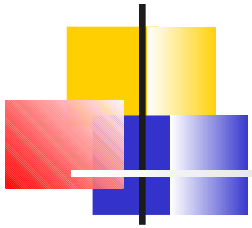
Open for *reading*

Returns NULL in case of error (e.g. File not found)



Opening a File for Reading

```
FILE* pFile;

pFile = fopen("test.c", "r");

if (pFile == NULL)
{
    printf ("Error opening file\n");
    return 1;
}

// if statement could be replaced by
// assert(pFile != NULL)
```



Windows and Slashes

Beware, Windows users:

If you try to open a file using an absolute path, like

```
fp = fopen("C:\Users\rtindell\silly.c","r");
```

you will get an error.

Why? Because the `\r` is interpreted as an escape character.

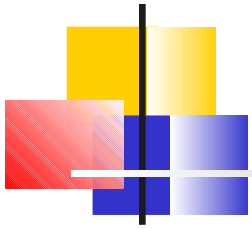
Two solutions: double your slashes:

```
fp = fopen("C:\\Users\\rtindell\\silly.c","r");
```

or use Unix-style separators:

```
fp = fopen("C:/Users/rtindell/silly.c","r");
```

The Windows-aware compiler will make the appropriate replacement.



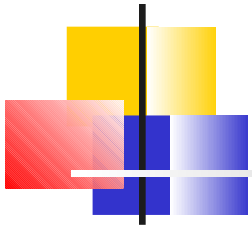
Function fgets()

To read a line of text from a file, use function fgets().

Similar to gets() *except*

- Reads from an open file rather than from the keyboard.
- Call specifies maximum number of chars to read (including null terminator.)
- If new line char terminates the read, it is included in the input area.

- We have used this before using the special file handle **stdin**



Function fgets()

```
char* fgets (char* buffer, int max, FILE* pFile)
```

Returns NULL in case of error
Returns buffer when successful.

address of input buffer

Maximum number of chars to read (including terminating null)

Pointer returned by fopen()

Reads to end of line, but no more than max-1 characters.
Newline character is included in the buffer (unlike gets.)
Null char terminator is added to chars read from file.

Function fgets()

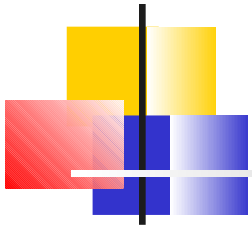
- Safer than gets() because you can specify the maximum number of chars to read.
- Can be used instead of gets() to read a line of text from the keyboard:

```
#define MAX_LINE 80
...
char input_buffer[MAX_LINE];
...
fgets (input_buffer, MAX_LINE, stdin);
```

Automatically defined FILE* for keyboard

Reading a File: test.c

```
#include <stdio.h>
#define MAX_LEN 1000

int main(void)
{
    char buffer[MAX_LEN];
    FILE* pFile;

    pFile = fopen("test.txt", "r");

    if (pFile == NULL)
    {
        printf ("Error opening file\n");
        return 1;
    }

    while ( fgets (buffer, MAX_LEN, pFile ) != NULL )
    {
        printf ("%s", buffer);
    }

    return 0;
}
```

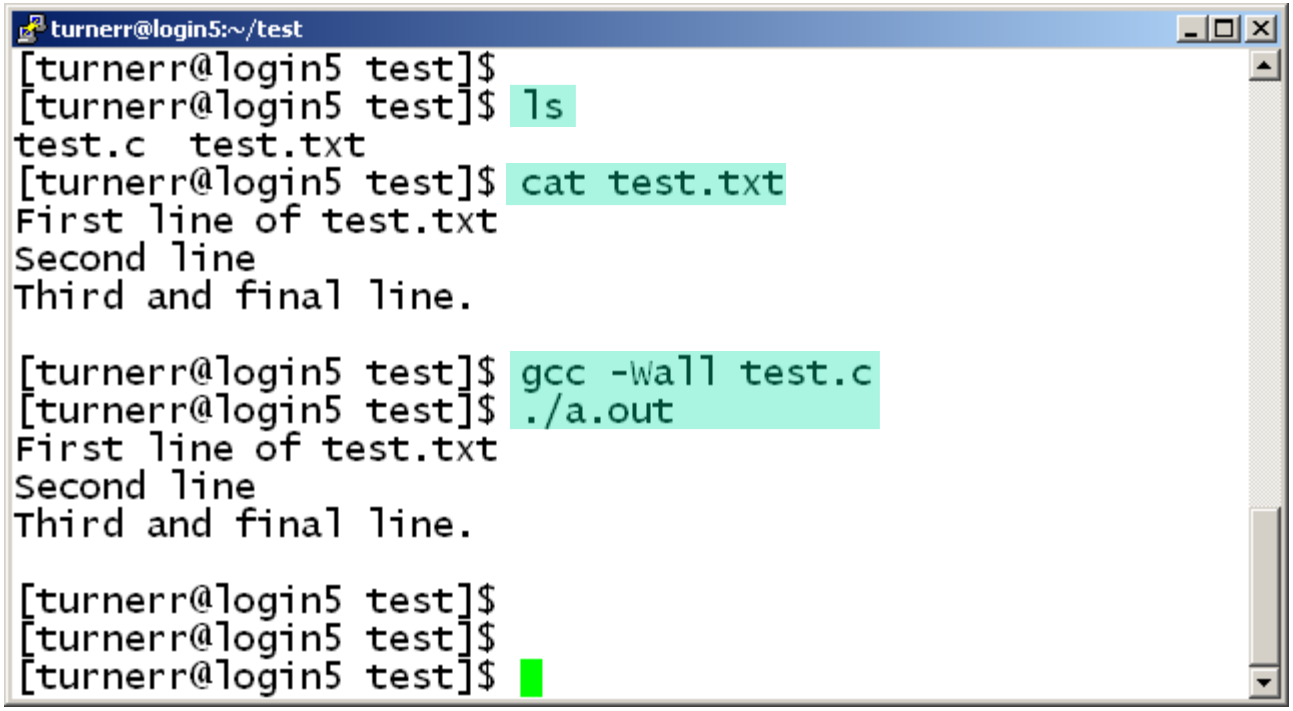


A File to Read

- Use your favorite editor to create a short text file called test.txt

```
First line of test.txt
Second line
Third and final line
```

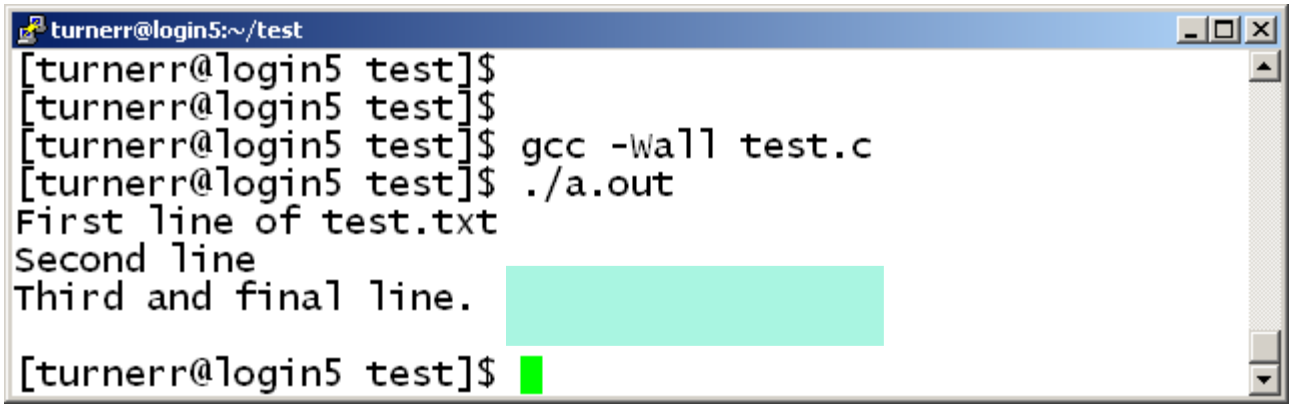
Compile and Run test.c



What if input is stopped by the limit?

```
#include <stdio.h>
#define MAX_LEN 10
int main()
{
    char buffer[MAX_LEN];
    FILE* pFile;
    pFile = fopen("test.txt", "r");

    if (pFile == NULL)
    {
        printf ("Error opening file\n");
        return 1;
    }
    while ( fgets (buffer, MAX_LEN, pFile ) != NULL )
    {
        printf ("%s", buffer);
    }
    return 0;
}
```

Output looks the same!

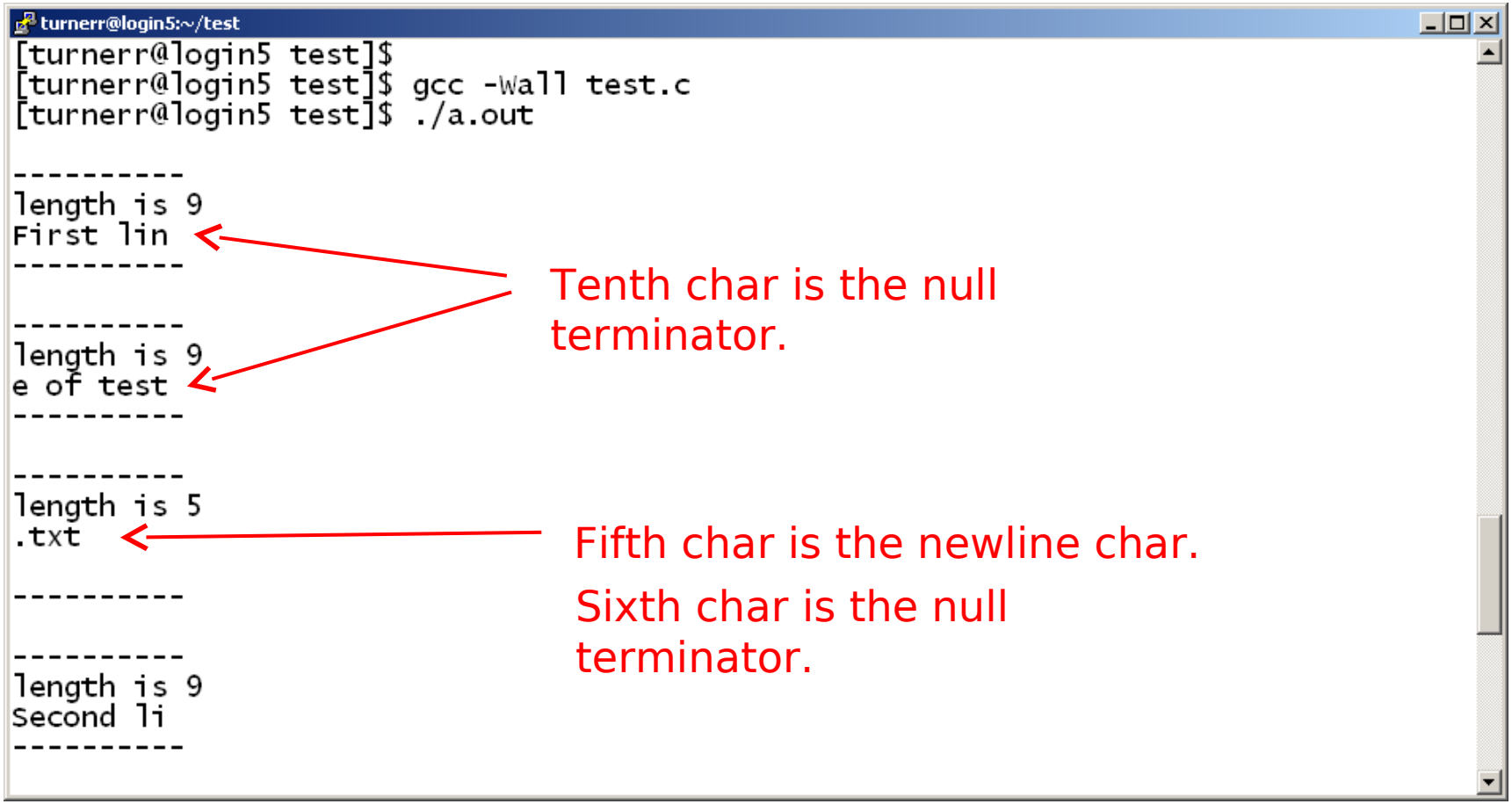


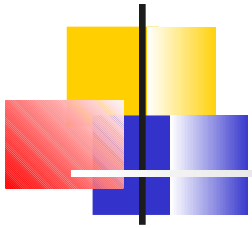
What's going on here?

Separate the outputs.

```
#include <string.h>
#include <stdio.h>
#define MAX_LEN 10

int main()
{
    char buffer[MAX_LEN];
    FILE* pFile;

    pFile = fopen("test.txt", "r");
    if (pFile == NULL)
    {
        printf ("Error opening file\n");
        return 1;
    }

    while (fgets (buffer, MAX_LEN, pFile) != NULL)
    {
        printf ("\n----------\n");
        printf ("length is %d\n", (int) strlen(buffer));
        printf ("%s", buffer);
        printf ("\n----------\n");
    }

    return 0;
}
```

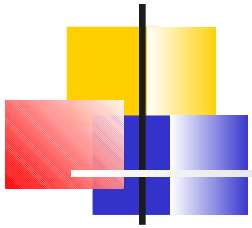
Separate the outputs.





Writing a File

- Use “w” as the mode in fopen() to write
  - Creates file if it does not exist.
  - Overwrites old version if file does exist.
- Use “a” to *append* to an existing file.
  - Creates file if it does not exist.



Closing a File

```
int fclose (FILE* pFile);
```

Undo the effects of fopen();

Writes any buffered data to the file.

Frees resources used by the file.

Done automatically when program ends.

Good practice to close files when finished.

Writing to a File

```
int fputs (char* s, FILE* pFile)
```

Returns EOF on error

Returns zero when successful

Pointer to text string to be written

Pointer returned by fopen()

EOF is a symbolic constant for an integer value defined in stdio.h

Usually -1 (But don't assume this value. Use the symbol EOF.)

Similar to puts()

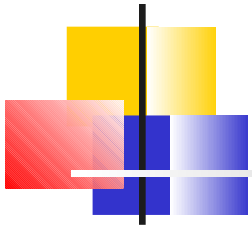
BUT ...

fputs() DOES NOT append "\n" to the line like puts() does.

fputs() DOES NOT output the null

Program puts.c

```
#include <stdio.h>
#define MAX_LEN 1000

int main()
{
    FILE* pFile;

    pFile = fopen("foo.txt", "w");

    if (pFile == NULL)
    {
        printf ("Error opening file\n");
        return 1;
    }

    fputs("Humpty Dumpty sat on a wall\n", pFile);
    fputs("Humpty Dumpty had a great fall\n", pFile);

    fclose(pFile);
    printf ("File foo.txt written\n");
    return 0;
}
```



Formatted Output to a File

- A “file” version of printf allows us to do formatted output to a file.

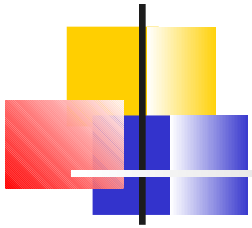
```
int fprintf (FILE* pFile, char* format, ...)
```

Returns EOF on error
Usually ignored

Pointer to FILE struct open for write

Format string (Same as for printf)

Variables to be formatted



Formatted File Input

- A “file” version of scanf allows us to read numbers, and other formatted information, from a file.

```
int fscanf (FILE* pFile, char* format, ...)
```

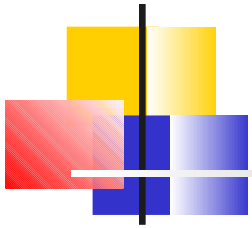
Returns number of variables filled,

or EOF on error

Pointer to open FILE struct

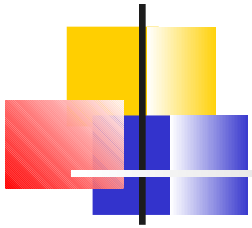
Format string (Same as for scanf)

Variables to read into



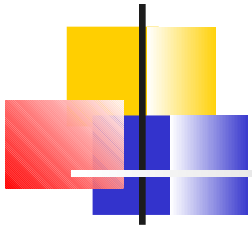
fscanf()

- Format string same as for scanf()
- Percent sign field specifies conversion.
- Space in the format says to skip over white space in the input file.
- Any character other than space or format specifier *must appear in the input file*
  - If present it is removed.
  - If not present results in an error.



int fgetc(FILE * f)

- Exact analogue of getchar()
- f must have already been opened for reading
- Gets the next character from the input buffer
- No interpretation of character done, no skipping of white space,
- Returns EOF when it has tried to read one position past the end of the file f.
- Why a return type of **`int`**?
- Since all 256 possible values of type char must be possible legitimate return values, EOF cannot be of type char, hence it must be of type int
- Assigning an int value to a char assigns the low order



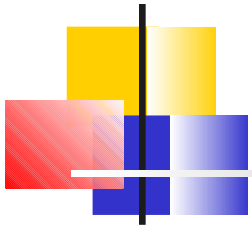
Checking for End of File

```
int feof (FILE* pFile)
```

Returns *true* when you have attempted to read beyond the end of the file.

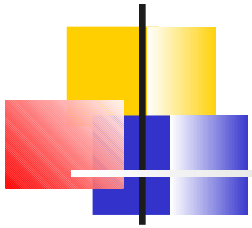
Returns false when end of file has not been reached.

Usage: Check *after* a read operation and *before* processing the input data.



fscanf()

- Format string same as for scanf()
- Percent sign field specifies conversion.
- Space in the format says to skip over white space in the input file.
- Any character other than space or format specifier *must appear in the input file*
  - If present it is removed.
  - If not present results in an error.



Example: copying a file

- The function **`fputc(f,char)`** outputs a character to a file.
- We next show a program that copies one text file to another.
- Note that we check **`feof`** after **`fgetc`** and before using the input of **`fgetc`**.

Program puts.c

```
void filecopy(FILE *dest, FILE *source)
  // Precondition: source open for
  //reading, dest open for writing
{
  char ch = fgetc(source);
  while( !feof(source))
  {
    fputc(ch,dest);
    ch = fgetc(source);
  }
}
```

If we declare ch to be of type int, we can use a test for EOF.

```
void filecopy(FILE *dest, FILE *source)
  // Precondition: source open for
  //reading, dest open for writing
{
  int ch;
  while((ch = fgetc(source)) != EOF)
  {
    fputc(ch,dest);
  }
}
```