UNIX and C Programming (COMP1000)

Lecture 6: Input and Output

Updated: 19th August, 2018

Department of Computing
Curtin University

Textbook Reading (Hanly and Koffman)

For more information, see the weekly reading list on Blackboard.

- **Chapter 11: Text and Binary File Processing**
  Consider reading Sections 10.1 and 10.2 (in Chapter 10)
  beforehand. These introduce structs, which are used in some
  of the examples in Chapter 11.

Outline

Intro to I/O

File I/O in C

Errors

Reading/Writing

Binary files

Standard Streams

Introduction to I/O

- ▶ Reading from and writing to files is a crucial part of any programming language.
- ▶ Virtually all programs need to do it (in the real world).
- ▶ Files provide permanent storage — "persistence" — unlike variables in memory.

Opening and Closing Files

- Before reading/writing a file, the file must be "opened".
- After reading/writing, the file must be "closed".
- If you're reading from a file, the file must exist.
- If you're writing to a file, you can choose whether:
    - the file will be created or overwritten, OR
    - the file will be appended to.

Streams

- Most input and output uses "streams".
- Characters go in one end and out the other — a queue.
- When you read a file:
  - Each character in the file is fed into the stream.
  - Your program reads and removes characters from the stream.
  - Often your program must wait for characters to become available.
- When you write to a file:
  - Your program feeds characters into the stream.
  - Characters are progressively removed from the stream and written to disk.
- A stream is created when you open a file.

Streams — Visualisation

Input stream | Output stream

file1.txt — — — — — → Program — — — — — → file2.txt
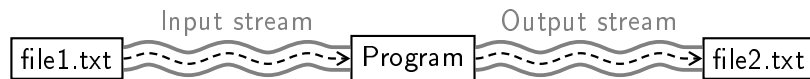
- ▶ Here, program reads from file1.txt and writes to file2.txt
- ▶ However, you can have as many streams as you like!
- ▶ Characters "flow" through each input stream *to* the program.
- ▶ More characters "flow" through each output stream *from* the program.

Buffering (1)

- ▶ File I/O is slow, compared to memory-based operations.
- ▶ Usually, characters to be read/written are stored temporarily in "buffers" (inside a stream).
- ▶ This allows characters to be read/written in large chunks — more efficient than one-at-a-time.
- ▶ Buffering often happens transparently, in the middle of a stream.

Buffering (2)

Input Buffering

- ▶ If you read one character from a file, the OS actually feeds a much larger chunk of the file into the stream.
- ▶ When you want the next character, it's already waiting in the buffer.

Output Buffering

- ▶ If you write one character to a file, the OS delays the actual write until enough characters accumulate, or the file is closed.
- ▶ If you *don't* close a file, you'll lose everything still in the output buffer.

Flushing Output Buffers

- ► Output buffers can be "flushed".
- ► This (also) happens automatically when a file is closed.
- ► Flushing an output buffer writes the output to disk immediately.

Seeking

- ▶ Normally a file is read/written sequentially.
- ▶ At any given moment, the stream "points" to a particular location in the file.
  - ▶ The first character on the first line is 0.
  - ▶ Each subsequent character is one greater than the previous.
- ▶ This is the location where characters will next be read or written.
- ▶ Jumping to another location (either forwards or backwards) is called "seeking".
- ▶ You must know the exact *byte* location.
- ▶ You cannot jump straight to a particular line number.

Man Pages (Manual Pages)

- You're about to be assaulted with many new C functions.
- The UNIX "man" utility can display documentation (as mentioned before).
- This command will display the man page for the strlen() function:

```
[user@pc]$ man 3 strlen
```

- Standard C functions are located in section 3 of the manual.
- You can omit the section number:

```
[user@pc]$ man strlen
```

(. . . but you may get a page from the wrong section!)

Man Page Example

```
STRLEN(3)        Linux Programmer's Manual        STRLEN(3)

NAME        strlen - calculate the length of a string

SYNOPSIS    #include <string.h>
        size_t strlen(const char *s);

DESCRIPTION
The strlen() function calculates the length of
the string s, not including the terminating '\0'
character.

RETURN VALUE
The strlen() function returns the number of
characters in s.
```

Man Page Information

As shown on the previous slide, each man page lists:

- NAME — The function name and purpose.
- SYNOPSIS —
  - The header file you must #include.
  - The function prototype/declaration.
- DESCRIPTION — A description of the function.
- RETURN VALUE — A description of the return value.

More information is also listed, including notes, bugs and related functions ("SEE ALSO").

Introduction to File I/O in C

- Uses the stdio.h library.
- File I/O is closely related to terminal I/O — printf() and scanf().
- Make good use of man pages!

FILE Pointers

- ▶ When you open a file in C you get a "FILE" pointer (FILE*).
- ▶ The FILE* is used to access the stream.
- ▶ When you read/write a open file (stream), you must supply the FILE*.

Note

- ▶ You never need to deal with the FILE type itself, only FILE*.
- ▶ You just pass the pointer around to various C functions.
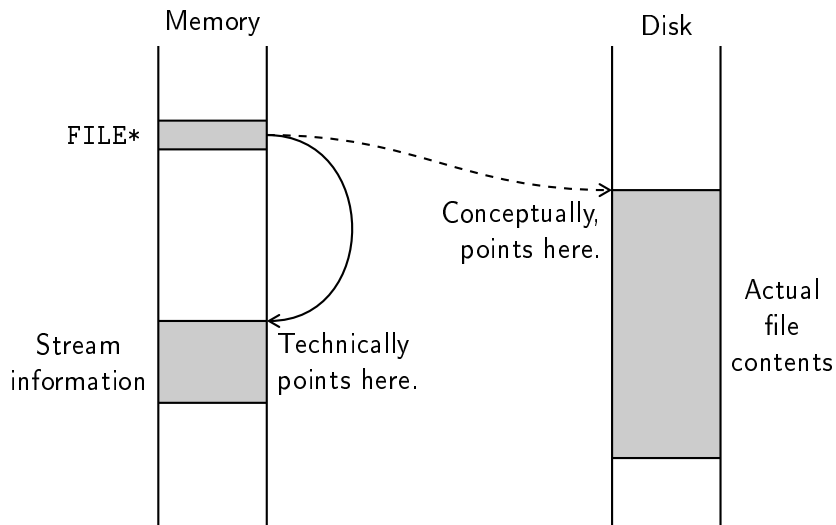
FILE Pointers — Concept and Reality

Reality

- ▶ A FILE* points to a FILE (of course) — a chunk of information in memory used to access a stream.
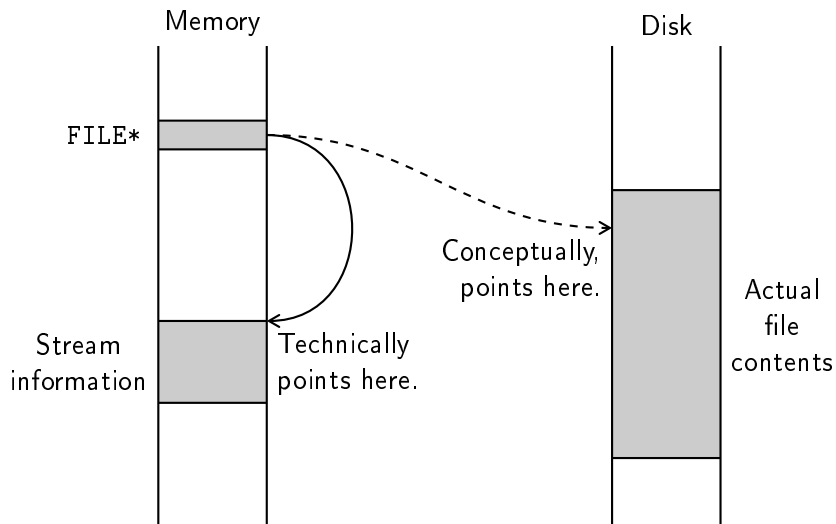- ▶ A FILE* is really just a normal pointer.

Concept

- ▶ You can *think* of a FILE* as a pointer to a place in a file.
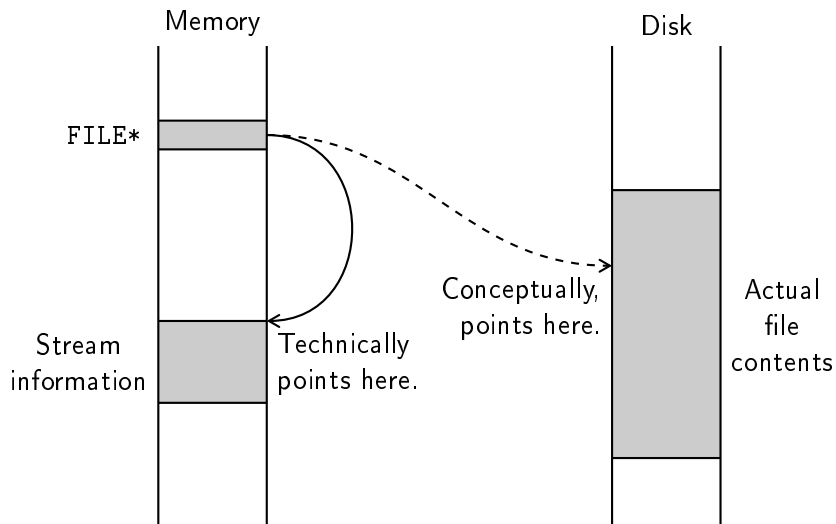- ▶ When you read/write, the pointer automatically moves forward.
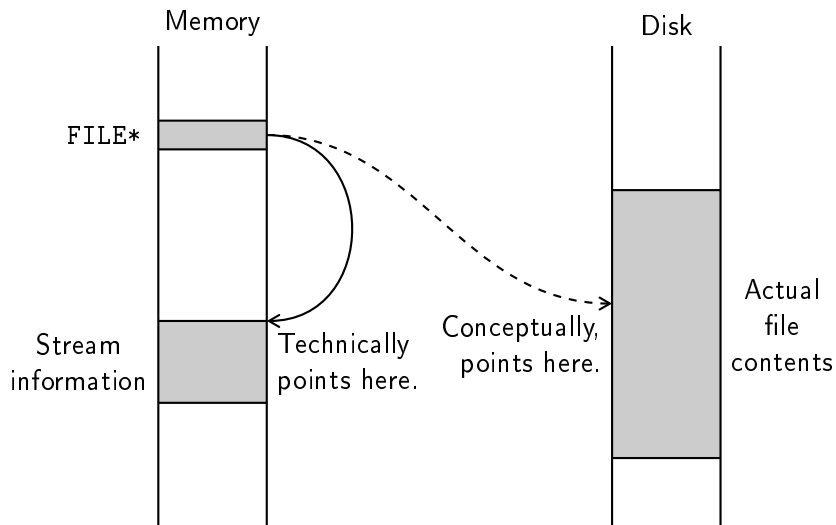
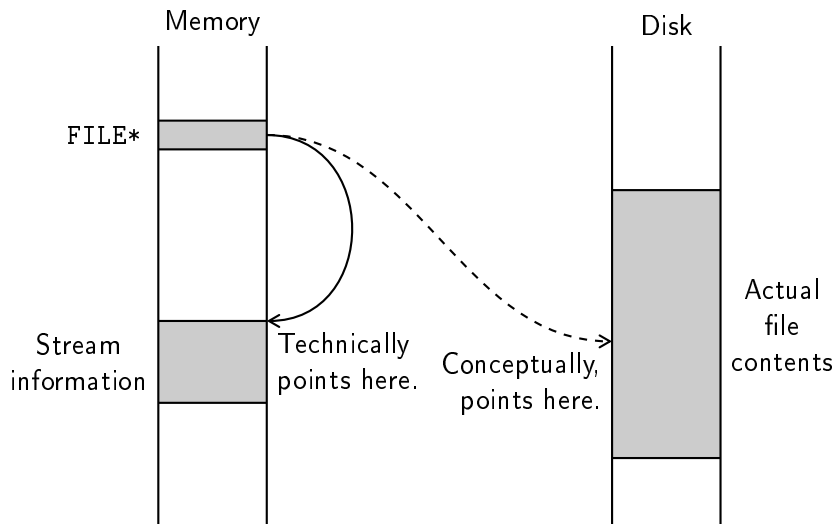## FILE Pointers — Visualisation

FILE Pointers — Visualisation

FILE Pointers — Visualisation

FILE Pointers — Visualisation

FILE Pointers — Visualisation

Opening and Closing Files in C

The fopen() function

- ▶ Opens a file.
- ▶ Takes two char* (string) parameters — a filename and a "mode" string.
- ▶ Returns FILE*, or NULL if the file couldn't be opened.

The fclose() function

- ▶ Closes a file.
- ▶ Takes a FILE* parameter.
- ▶ Returns an int indicating whether an error occurred.

Opening and Closing Files — Example

```
#include <stdio.h>

...

FILE* f;
f = fopen("filename.txt", "r");

... /* Read from the file */

fclose(f);
```

(Be careful — we haven't checked for errors here!)

File Modes

- ▶ The second parameter to fopen() is the "mode" string.
- ▶ Indicates what you want to do with the file.
- ▶ Always a *string*, even if only 1 character long.

Basic modes

"r"   read from a file (starting at the beginning)
"w"   write to a file (create or overwrite the file)
"a"   write to a file (starting at the end — "appending")

"Update" Modes

- Adding a "+" to the mode allows both reading and writing.
- This makes the stream bi-directional.
- You can switch between reading and writing (but you can't do both simultaneously).
- Going from reading to writing, you must perform a seek.
- Going from writing to reading, you must perform a seek *or* flush.

"Update" modes

`"r+"`	read & write (starting at the beginning)
`"w+"`	read & write (create or overwrite the file)
`"a+"`	read & write (starting at the end)

Text and Binary Modes

- Microsoft Windows distinguishes between "text" and "binary" modes.
- In "binary mode", the file is read/written as-is.
- In "text mode", Windows fiddles with some special characters.
- **Text mode is the default, but will corrupt binary files (under Windows)!**
- To select binary mode, place "b" in the mode string (e.g. `"rb"`, `"wb"`).
- UNIX (e.g. Linux and OS X) does not need this — the "b" flag is accepted but ignored.

Errors

- ► File I/O is error-prone.
- ► Possible errors include:
  - ► Trying to read a file that doesn't exist.
  - ► Trying to write to a file when the disk is full.
  - ► Trying to read/write to a file when you don't have permission.
  - ► Disk hardware errors.
- ► I/O errors are not necessarily your fault (though they might be).

Error Checking and Handling

- ▶ Your program should detect errors and handle them gracefully.
- ▶ Newer languages (like Java) deal with I/O errors by throwing "exceptions", which can be caught and handled.
- ▶ C leaves it up to you to *check* when an error occurs.
- ▶ Output helpful error messages.

Error Checking: Opening Files

- Recall that fopen() returns NULL if a file can't be opened.
- You should *check* for this (with an if statement):

```c
FILE* f = fopen("file.txt", "r");
if(f == NULL) {
    printf("Error: could not open 'file.txt'\n");
}
else {
    ... /* Read from the file */

    fclose(f);
}
```

Error Handling — `perror()`

- Don't leave the user to figure out what went wrong!
- The `perror()` function determines what the error was and prints out a relevant message.
- Takes a `char*` — a prefix to the error message.
- Returns `void`, and prints out the parameter plus the error.

```
FILE* f = fopen("file.txt", "r");
if(f == NULL) {
    perror("Error opening 'file.txt'");
}
else { ... /* Read and close the file */ }
```

Possible output (say `file.txt` does not exist):

```
Error opening 'file.txt': No such file or directory
```

More Error Checking — `ferror()`

- The `ferror()` function checks whether an error has occurred.
- Takes a `FILE*` parameter (must be non-NULL).
- Returns an `int` — zero (no error) or non-zero (error).

```c
FILE* f = fopen("file.txt", "r");
if(f == NULL) { ... }
else {
    ... /* Read from the file */

    if(ferror(f)) {
        perror("Error reading from 'file.txt'\n");
    }
    fclose(f);
}
```

Reading and Writing

- Reading usually uses mode `"r"`.
- Writing usually uses mode `"w"`.
- Every read or write will move the FILE pointer forward.
- However, there are different functions to read/write data, based on the **type** of data:
  - Strings with embedded values.
  - Individual characters.
  - Lines of text.
  - Binary data.
- You may also need different algorithms, based on the file format.
- You may (or may not!) know how many data elements are stored in the file.

Checking for the End of File (EOF)

- ▶ How big is the file? You don't always know.
  - ▶ (You can find the number of bytes, but each data element may occupy an unknown number of bytes.)
- ▶ The read functions – `fscanf`, `fgets`, `fgetc` – will all tell you when the file has ended.
- ▶ That's the trick: you *won't know* until *after* you try to read past the end of file.
  - ▶ The last read operation always fails (unless you know in advance how many reads you can do).

Basic File Reading Algorithm

```
int done = FALSE;
do {
    Attempt a read operation
    if(the read succeeded)
        Store the data (e.g. in an array)
    else
        done = TRUE;
}
while(!done);

if(ferror(the file pointer))
    Handle error and de-allocate any stored data
else
    Success
```

fprintf() and fscanf()

- Like printf() and scanf(), but used with files.
- Both take an extra FILE* parameter (first).

fprintf() example

```
FILE* f = fopen("output.txt", "w");
int number;
...
fprintf(f, "The number is %d\n", number);
```

fscanf() continued

- fscanf() returns *either*:
  - the number of items successfully read, OR
  - the preprocessor constant EOF (end-of-file), but <u>only if</u> it doesn't read anything first.
- Compare the return value to the number you expected:

```
FILE* f = fopen("input.txt", "r");
int x, y, z, nRead;
...                          3 items
nRead = fscanf(f, "%d %d %d", &x, &y, &z);
if(nRead != 3)
    /* An error or end of file occurred. */
```

Note on EOF

- EOF is a preprocessor constant, guaranteed to be *some* negative integer.

fputc() and fgetc()

- fputc() writes one character to an output stream.
- fgetc() reads one character from an input stream.

fputc() example

```
FILE* f = fopen("output.txt", "w");
char ch;
...
fputc(ch, f);
```

fgetc() continued

- fgetc() returns an int (not a char) -- why?
- Because, if it fails, it returns EOF.
- EOF is a negative integer, not representable as a char value.

```c
FILE* f = fopen("input.txt", "r");
int ch;
...
ch = fgetc(f);
if(ch == EOF)
    /* Error or end-of-file. */
else
    /* Success -- typecast ch to a char. */
```

Writing a line of text — fputs()

- fputs() writes a string, plus a new line ('\n').
- Takes two parameters: char* and FILE*.

Example

```
FILE* f = fopen("output.txt", "w");
char* str = "Hello world";
...
fputs(str, f);
```

(Here, the FILE* parameter comes *last*.)

Reading a line of text — `fgets()`

- `fgets()` reads a string, taking 3 parameters:
  - `char*` — an array to store the text.
  - `int` — the size of the array.
  - `FILE*` — an input stream.
- Stops at the next newline, or the size of the array *minus 1* (whichever comes first).

```c
#define INPUT_SIZE 21
...
FILE* f = fopen("input.txt", "r");
char str[INPUT_SIZE];
...
if(fgets(str, INPUT_SIZE, f) == NULL)
    /* Error or end-of-file. */
else
    /* Success -- you can safely use str. */
```

Other Reading/Writing Functions

These functions are all essentially redundant:

getc() and putc()

Reads/writes a character (like fputc() but with possible side-effects).

getchar() and putchar()

Reads/writes a character from/to the terminal.

gets() and puts()

Reads/writes a line of text from/to the terminal.

Warning

gets() **should never be used**, because there's no way to prevent buffer overflows. Use fgets() instead.

Multiple Files Example

This copies "input.txt" to "output.txt", inserting dashes:

```c
FILE *inFile = fopen("input.txt", "r");
FILE *outFile = fopen("output.txt", "w");
int ch;
... /* Error checking */
do {
    ch = fgetc(inFile);
    if(ch != EOF) {
        fputc((char)ch, outFile);
        fputc('-', outFile);
    }
} while(ch != EOF);
... /* More error checking */
fclose(inFile);
fclose(outFile);
```

Flushing — `fflush()`

- An output stream can be flushed with `fflush()`.
- This may be useful in programs that run for a very long time:
  - number crunching
  - event logging
  - databases
- Flushing an output stream can help prevent data loss (if the program, OS or computer crashes).

Binary files

- ▶ Many files are not human-readable — these are often called "binary" files...
  - ▶ ...even though all data in computing is binary!
- ▶ All files are made up of bytes — 8-bit integers.
- ▶ In a text file, bytes represent characters — printable symbols. Characters in turn form words, numbers, etc.
- ▶ In a binary file, bytes *directly* represent integers, real numbers, etc. They do not (generally) represent characters.

Binary file formats

- There are no words, lines or paragraphs in a binary file.
- Each data item occupies a fixed number of bytes.
- ints, doubles, etc. are stored as they would be in memory.
    - A 32-bit int always occupies 32 bits, i.e. 4 bytes (remember sizeof?).
    - Compare this to text files, where a 32-bit int could be 1 byte (e.g. "2") or 10 bytes (e.g. "2000000000").
- Due to fixed sizes, there are no spaces or other delimiters around data items — no need to separate them.
- This makes it impossible to distinguish between them, unless you know the precise format in advance.

Reading and Writing Binary Data

- ▶ The fread() and fwrite() functions deal with binary data.
- ▶ (Under Windows, you'll need the "rb" or "wb" modes.)
- ▶ When writing to a binary file, no conversion is done.
  - ▶ Before writing the int 123 to a text file, it must be converted to the string "123" (i.e. a sequence of digit characters).

Writing binary data — fwrite()

- Writes an array of data (of any type).
- Takes four parameters:
  - void* — a pointer to the data to write.
  - int — the size of each data element to write.
  - int — the total number of elements to write.
  - FILE* — an output stream.

```
#define LENGTH 5
...
int data[LENGTH] = {10, 20, 30, 40, 50};
FILE* f = fopen("output.txt", "wb");
...
fwrite(data, sizeof(int), LENGTH, f);
```

(Note: "sizeof(int)" gives you the number of bytes in an int.)

Reading binary data — `fread()`

- Reads an array of data (of any type).
- Takes the same four parameters as `fwrite()`:
    - `void*` — a pointer to an array to read into.
    - `int` — the size of each data element to read.
    - `int` — the *maximum* number of elements to read.
    - `FILE*` — an input stream.
- Returns the number of elements *actually* read (like `fscanf`).

```c
#define MAXLEN 5
...
int data[MAXLEN], length;
FILE* f = fopen("input.txt", "rb");
...
length = fread(data, sizeof(int), MAXLEN, f);
if(length < MAXLEN) ... /* EOF, error or success? */
```

Seeking — `fseek()`

- Can position the FILE pointer anywhere in the file.
- Mostly useful with binary files — you know where something ought to be.
- Can be used in three ways:
    - Move the file pointer f to 15 bytes after the start of the file:

    ```
    fseek(f, 15, SEEK_SET);
    ```

    - Moves the file pointer f 15 bytes back from its current location:

    ```
    fseek(f, -15, SEEK_CUR);
    ```

    - Moves the file pointer f to 15 bytes before the end of the file:

    ```
    fseek(f, -15, SEEK_END);
    ```

Seeking — rewind() and ftell()

rewind()

- ▶ Resets the FILE pointer to the start of the file.
- ▶ This may be useful when you need to read a file multiple times.
- ▶ rewind(f) is equivalent to fseek(f, 0, SEEK_SET).

ftell()

- ▶ Reports the current location within a file.
- ▶ Takes a FILE*.
- ▶ Returns the location as a long (indicating the number of bytes from the start of the file).

Standard Streams

- ▶ Not all streams are connected to a file
- ▶ In C, there are three special, pre-defined FILE pointers:
  - ▶ stdin ("standard input") — reads from the terminal
  - ▶ stdout ("standard output") — writes to the terminal
  - ▶ stderr ("standard error") — also writes to the terminal
- ▶ These do not need to be opened or closed.
- ▶ fprintf(stdout, ...) is equivalent to printf(...)
- ▶ fscanf(stdin, ...) is equivalent to scanf(...)

Standard Streams — Visualisation
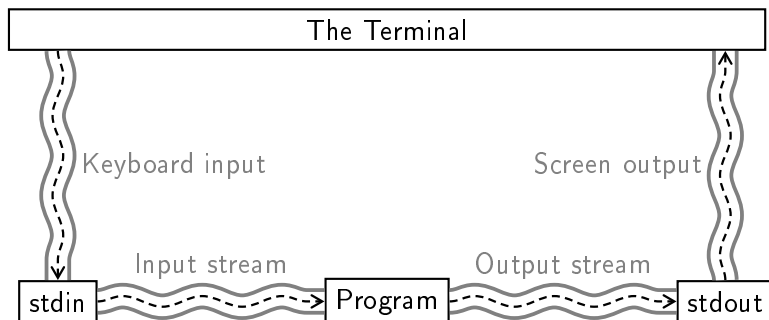
- ▶ To the program, stdin and stdout look like ordinary files. . .
- ▶ . . . except they're actually connected to the terminal, not to the disk.

Redirection

▶ The UNIX shell (sh, csh, bash, etc.) can "redirect" the
  standard streams — most commonly stdin and stdout:

▶ Say you normally run "program" with parameters "...":

```
[user@pc]$ ./program ...
```

▶ Standard output can be redirected to a file, instead of the
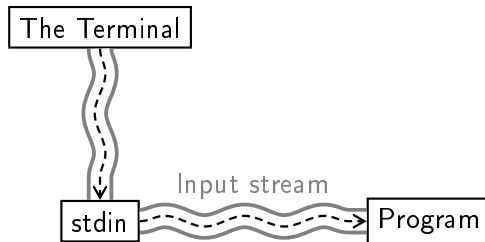 terminal:

```
[user@pc]$ ./program ...   >output.txt
```

▶ Standard input can be redirected so that it comes from a file:

```
[user@pc]$ ./program ...   <input.txt
```
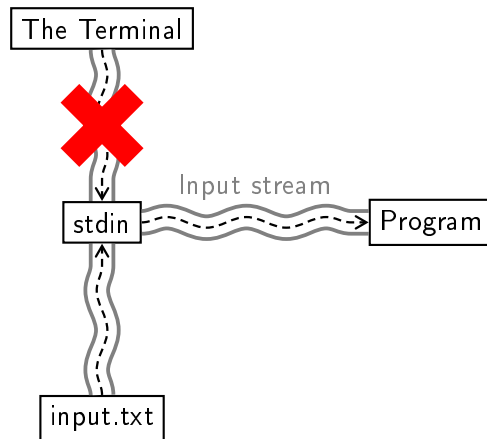
▶ You can also do both at once:

```
[user@pc]$ ./program ...   <input.txt >output.txt
```

Redirecting Stdin — Visualisation

- This is standard input, *un*redirected (i.e. as normal)

Redirecting Stdin — Visualisation

- Stdin has been redirected from the terminal to input.txt
- The program doesn't know that stdin has changed

Piping

- ▶ Piping is a special form of redirection.
- ▶ The UNIX shell can "pipe" the output of one program to the input of another.
- ▶ On the command-line:

  ```
  [user@pc]$ ./program1 ...   | ./program2 ...
  ```

  (The "|" symbol is called the "pipe" character.)
- ▶ This will cause both program1 and program2 to run simultaneously
- ▶ Whatever program1 outputs (e.g. with printf()), program2 can read in (e.g. with scanf())
- ▶ Most UNIX commands are designed with this in mind.

Piping — Visualisation
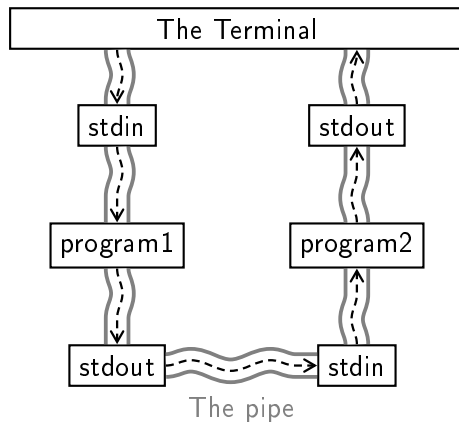
The pipe

- ▶ Program1 sends data to program2 via the pipe
- ▶ Neither program (necessarily) realises this is happening

Coming Up

- ▶ The next lecture will look at structs and linked lists.