What have we learned?

- `fopen` returns a FILE pointer: (null pointer if fails)

```
fp = fopen("in.dat", "r");
```

- The `fclose` function closes a file that is no longer in use:

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

Text

00110011	00110010	00110111	00110110	00110111
'3'	'2'	'7'	'6'	'7'

Binary

01111111	11111111

Block I/O

- `fread` and `fwrite` allow a program to read and write large blocks of data in a single step

- `fread` and `fwrite` are used primarily with binary streams, although—with care—it's possible to use them with text streams

`fwrite` function

- `fwrite` copies an array from memory to a stream

array

size of each element (in bytes)

```
fwrite(a, sizeof(a[0]),
       sizeof(a) / sizeof(a[0]), fp);
```

number of elements to be copied

FILE pointer

- returns the number of elements actually written

`fread` function

- `fread` will read the elements of an array from a stream

array

size of each element (in bytes)

```
n = fread(a, sizeof(a[0]),
          sizeof(a) / sizeof(a[0]), fp);
```

number of elements to read

FILE pointer

- returns the number of elements actually read

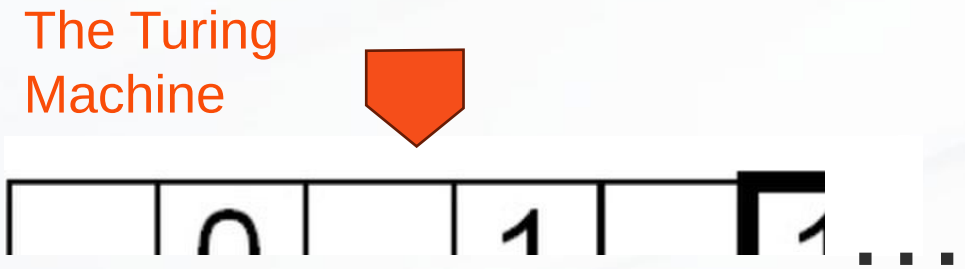
Block I/O

- `fwrite` is convenient to save data before terminating a program.
- Later, the program (or another program) can use `fread` to read the data back into memory.
- The data doesn't have to be in array form.
- A call of `fwrite` that writes a variable/structure/union… `s` to a file:

```
fwrite(&s, sizeof(s), 1, fp);
```

File positioning

- Every stream has an associated ***file position.***
- When a file is opened, the file position is set at the beginning of the file.
  - In “append” mode, the initial file position may be at the beginning or end, depending on the implementation.
- When a read or write operation is performed, the file position advances automatically, providing sequential access to data.



File positioning

- `fseek` function changes the file position associated with the first argument (a FILE pointer).
- The third argument is one of three macros:

`SEEK_SET`	Beginning of file
`SEEK_CUR`	Current file position
`SEEK_END`	End of file

- The second argument, which has type `long int`, is a (possibly negative) byte count, to offset from any of the three positions.

File positioning examples

- Using `fseek` to move to the beginning of a file:

```
fseek(fp, 0L, SEEK_SET);
```

- Using `fseek` to move to the end of a file:

```
fseek(fp, 0L, SEEK_END);
```

- Using `fseek` to move back 10 bytes:

```
fseek(fp, -10L, SEEK_CUR);
```

- If an error occurs (the requested position doesn't exist, for example), `fseek` returns a nonzero value.

File positioning

- The file-positioning functions are best used with binary streams.
- C doesn't prohibit programs from using them with text streams, but certain restrictions apply.
- For text streams, `fseek` can be used only to move to the beginning or end of a text stream or to return to a place that was visited previously.

To remember a position / "bookmark"

- The `ftell` function returns the current file position as a long integer.
- The value returned by `ftell` may be saved and later supplied to a call of `fseek`:

```
long file_pos;
…
file_pos = ftell(fp);
   /* saves current position */
…
fseek(fp, file_pos, SEEK_SET);
   /* returns to old position */
```

`rewind` function

- `rewind` function sets the file position at the beginning.

- The call `rewind(fp)` is nearly equivalent to `fseek(fp, 0L, SEEK_SET)`.

Example: Copying a file

- The program makes a copy of a file.
- The names of the original and new files will be specified on *the command line*.
- An example that uses `fcopy` to copy the file `f1.c` to `f2.c`:

```
fcopy f1.c f2.c
```

- `fcopy` will issue an error message if there aren't exactly two file names on the command line or if either file can't be opened.

Obtaining file names from the command line

- To access command-line arguments by defining `main` as a function with two parameters:

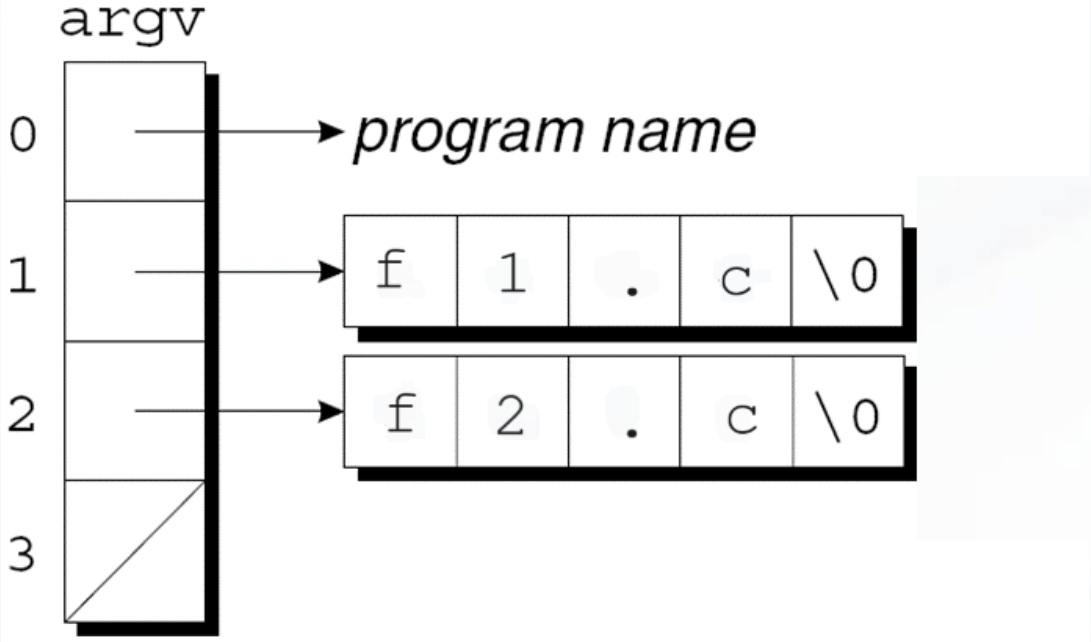
```
int main(int argc, char *argv[])
{
   …
}
```

- `argc` is the number of command-line arguments (including the program name)
- `argv` is an array of pointers to the argument strings.

Obtaining file names from the command line

- `argv[0]` points to the program name
  `argv[1]` through `argv[argc-1]` point to the remaining arguments
  `argv[argc]` is a null pointer

```
fcopy f1.c f2.c
```



```c
#include <stdio.h>
#include <stdlib.h>
#define BUFFSIZE 5000000
char buff[BUFFSIZE];
int main(int argc, char *argv[])
{
    FILE *source_fp, *dest_fp;
    if (argc != 3) {
        fprintf(stderr, "usage: fcopy source dest\n");
        exit(EXIT_FAILURE);
    }
    if ((source_fp = fopen(argv[1], "rb")) == NULL) {
        fprintf(stderr, "Can't open %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }
    if ((dest_fp = fopen(argv[2], "wb")) == NULL) {
        fprintf(stderr, "Can't open %s\n", argv[2]);
        fclose(source_fp);
        exit(EXIT_FAILURE);
    }
```

```
    fseek(source_fp, 0L, SEEK_END);
    long int file_length = ftell(source_fp);
    if(file_length>BUFFSIZE) {
        fprintf(stderr, "File too large to copy!\n");
        fclose(source_fp);
        fclose(dest_fp);
        exit(EXIT_FAILURE);
    };

    rewind(source_fp);
    fread(buff, sizeof(char), file_length, source_fp);
    fwrite(buff, sizeof(char), file_length, dest_fp);

    fclose(source_fp);
    fclose(dest_fp);
    return 0;
}
```

Formatted I/O

- `printf` and related functions convert data from *binary* form to *text* form during output.

variable number of arguments

```
int fprintf(FILE *stream,
            const char *format, ...);
```

return the number of characters written (negative for errors)

- `scanf` and related functions convert data from *text* form to *binary* form during input.

```
int fscanf(FILE *stream,
           const char *format, ...);
```

returns the number of input items successfully assigned

Formatted I/O

- A call of `printf` / `scanf` is equivalent to a call of `fprintf` / `fscanf` with `stdout` / `stdin` as the first argument.

```
fprintf(stdout, …)
fscanf(stdin, …)
```

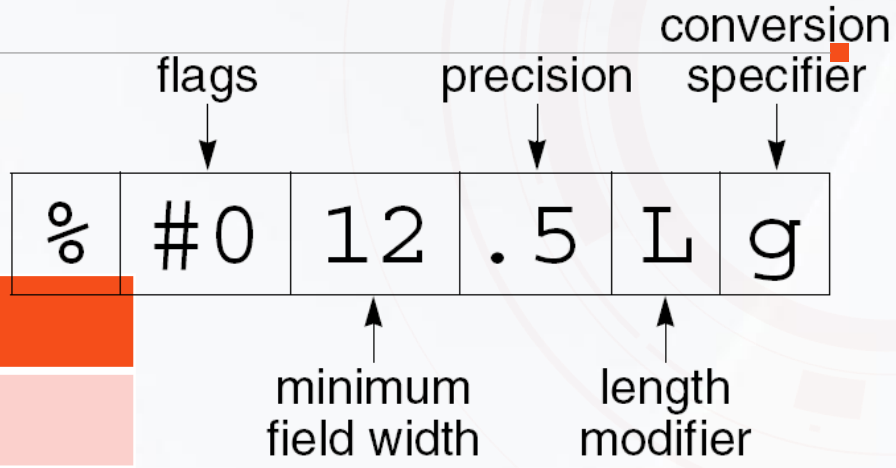
⟷

```
printf(…)
scanf(…)
```

- Print to `stderr` for error messages to be shown on the screen (even when `stdout` is redirected)

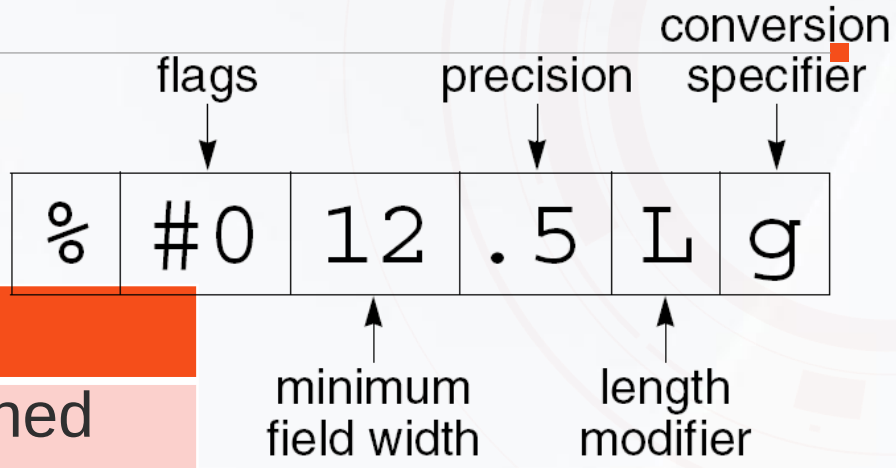
```
fprintf(stderr, "Error: data file can't be opened.\n");
```

Conversion specifiers



specifier	Meaning
`d`, `i`	an `int` value in decimal form.
`o`, `u`, `x`, `X`	an `unsigned int` value in base 8 (`o`), base 10 (`u`), or base 16 (`x`, `X`) form. `x`/`X` displays digits `A-F` in lower/upper case.
`f`	a `double` value in decimal form, putting the decimal point in the correct position. 6 digits after decimal point by default.
`e`, `E`	a `double` value to scientific notation. 6 digits after decimal point by default.
`g`, `G`	`g`/`G` converts a `double` value to either `f`/`F` form or `e`/`E` form.

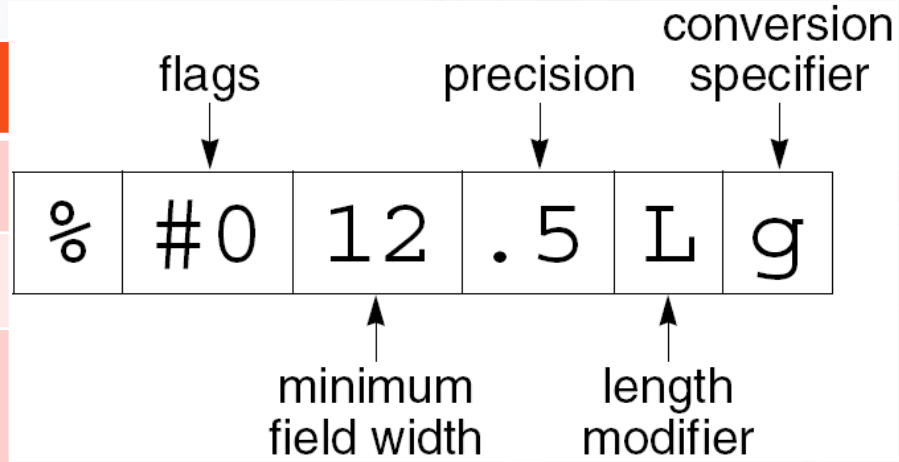
Conversion specifiers



specifier	Meaning
`c`	Displays an `int` value as an unsigned character
`s`	Writes the characters pointed to by the argument. Stops writing when the number of bytes specified by the precision (if present) is reached or a null character is encountered.
`p`	Print the address in hexadecimal format
`n`	Stores (no output) the number of characters written so far by this call
`%`	Writes the character `%`.

…`printf` Format Conversion Specifications

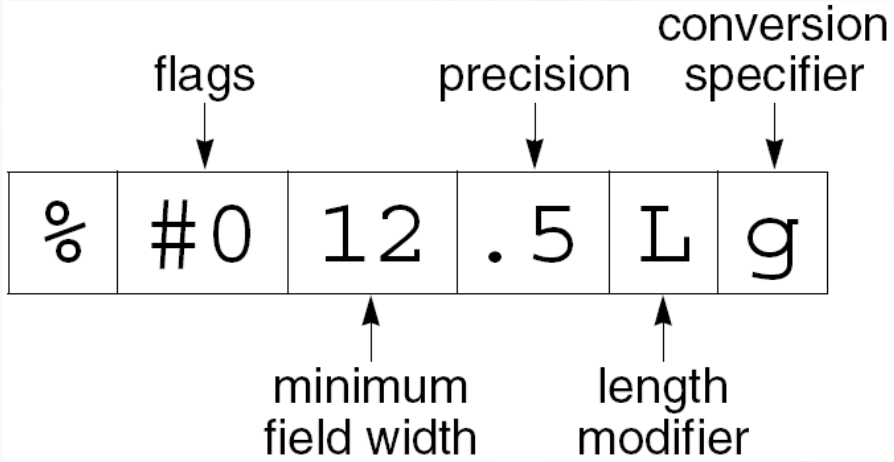
Flags	Meaning
-	Left-justify
+	Signed values always showing +/- sign
space	None negative numbers preceded by as space
#	Octal number begin with 0, hexadecimal numbers begin with 0x or 0X, Floating-point numbers always show decimal points. Trailing zeros aren't removed from numbers printed with the `g` or `G` conversions
0	Add leading zeros up to the field width



More than one flag permitted

…`printf` Format Conversion Specifications

- ***Minimum field width*** (optional). A shorter item will be padded.
  - By default, spaces are added to the left of the item.
- A longer item will be displayed completely.
- The field width is either an integer or the character `*`.
  - If `*` is present, the field width is obtained from the next argument.



Examples of …`printf` with `%d` conversion

Specification	Result for 123	Result for -123
%8d	•••••123	••••-123
%-8d	123•••••	-123••••
%+8d	••••+123	••••-123
% 8d	•••••123	••••-123
%- 8d	•123••••	-123••••
%-+8d	+123••••	-123••••
%08d	00000123	-0000123
%+08d	+0000123	-0000123
% 08d	•0000123	-0000123

- indicates a space

Effect of the # flag

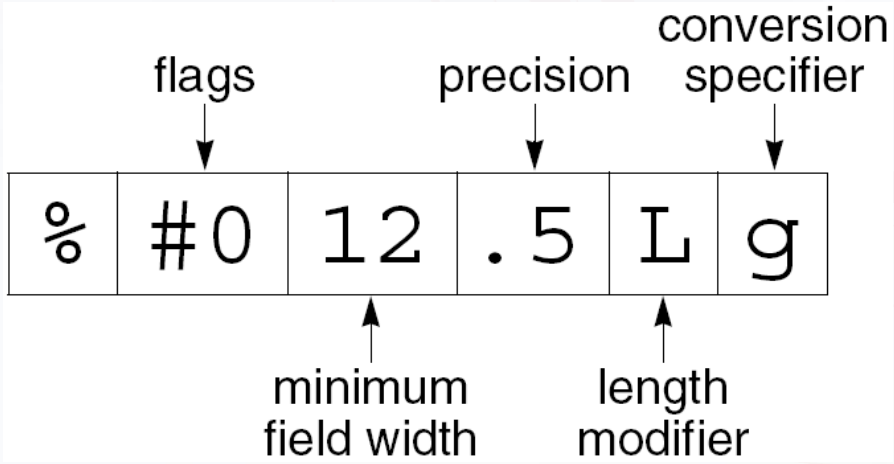
Specification	Result for 123	Result for 123.0
%8o	•••••173	
%#8o	••••0173	
%8x	••••••7b	
%#8x	••••0x7b	
%8X	••••••7B	
%#8X	••••0X7B	
%8g		•••••123
%#8g		•123.000
%8G		•••••123
%#8G		•123.000

- • indicates a space

Examples of the %g conversion

Number	Result of applying %.4g
123456.	1.235e+05
12345.6	1.235e+04
1234.56	1235
123.456	123.5
12.3456	12.35
1.23456	1.235
.123456	0.1235
.0123456	0.01235
.00123456	0.001235
.000123456	0.0001235
.0000123456	1.235e-05
.00000123456	1.235e-06

…`printf` Format Conversion Specifications



- ***Precision*** (optional). Meaning depends on the conversion:

  `d`, `i`, `o`, `u`, `x`, `X`: minimum number of digits (leading zeros added if the number has fewer digits)

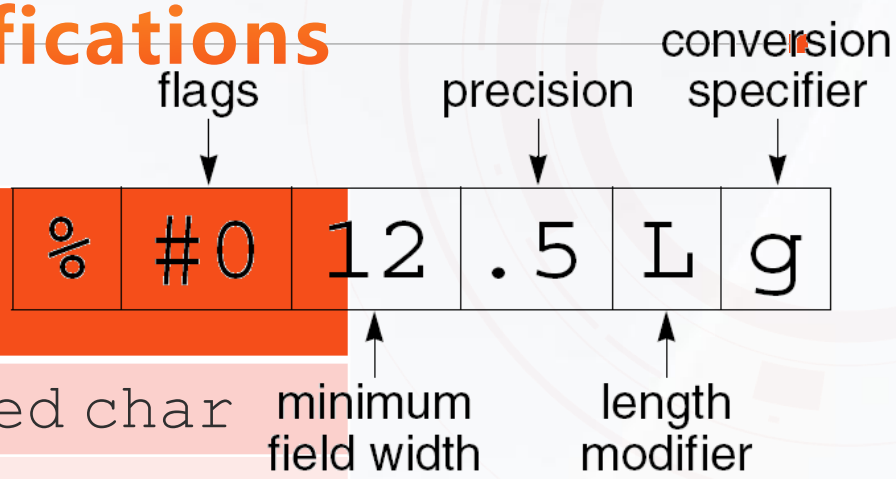
  `a`, `A`, `e`, `E`, `f`, `F`: number of digits after the decimal point

  `g`, `G`: number of significant digits

  `s`: maximum number of bytes

- The precision is a period (`.`) followed by an integer or the character `*`.
  - If `*` is present, the precision is obtained from the next argument.

…printf Format Conversion Specifications



Length modifier	Conversion Specifiers	Meaning
hh	d, i, o, u, x, X	signed char, unsigned char
h	d, i, o, u, x, X	short int, unsigned short int
l	d, i, o, u, x, X	long int, unsigned long int
	c	wint_t
	s	wchar_t *
ll	d, i, o, u, x, X	long long int, unsigned long long int
L	a, A, e, E, f, F, g, G	long double

The …`scanf` Functions

- `fscanf` reads data items from an input stream, using a format string to indicate the input format.
- After the format string, any number of pointers follow as additional arguments—each pointing to an input item.
- Input items are converted (according to the format string) and stored in these objects.

`scanf(…)` ⟺ `fscanf(stdin, …)`

Character I/O

- Library functions read and write single characters (bytes).
- Work with both text and binary streams.
- The functions treat characters as values of type `int`, not `char`.
- One reason is that the input functions indicate an end-of-file (or error) condition by returning `EOF`, which is a negative integer constant.

Output functions

- `fputc` and `putc` write a character to a stream:

```
fputc(ch, fp);   /* writes ch to fp */
putc(ch, fp);
```

Usually a macro

- `putchar` writes one character to `stdout`:

```
putchar(ch);   ⟺   fputc(ch, stdout);
```

Usually a macro

- All three functions set the error indicator for the stream and return `EOF`.
- Otherwise, they return the character that was written.

Other input functions

- `fgetc` and `getc` read a character from a stream:

```
ch = fgetc(fp);
ch = getc(fp);
```

Usually a macro

- All three functions treat the character as an `unsigned char` value (which is then converted to `int` type before it's returned).
- Store the return value in an `int` variable, not a `char`!
- As a result, they never return a negative value other than `EOF`.

`getchar()` ⟷ `fgetc(stdin)`

Usually a macro