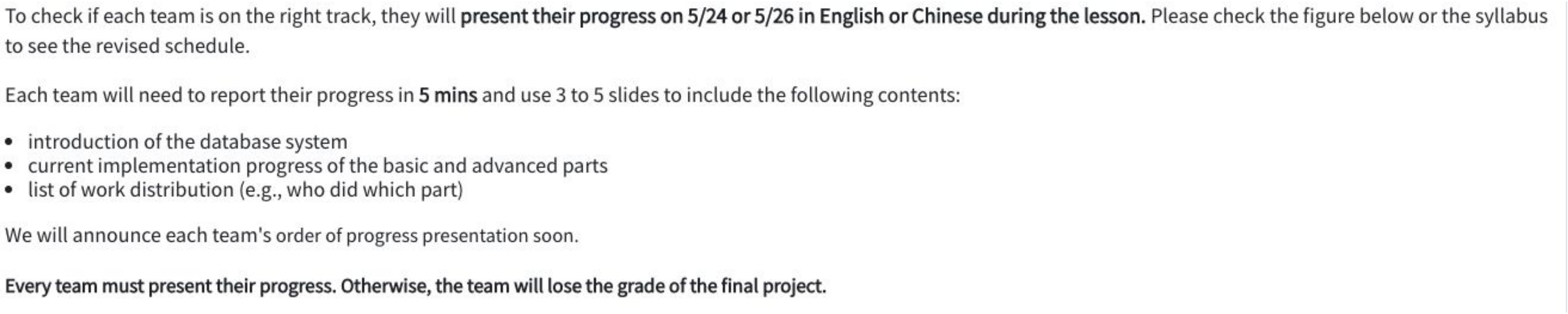
Input/Output (2)

Program Design (II)

2022 Spring

Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

All materials are provided by C Programming: A Modern Approach by K. N. King, 2nd edition, 2008, W. W. Norton & Company.



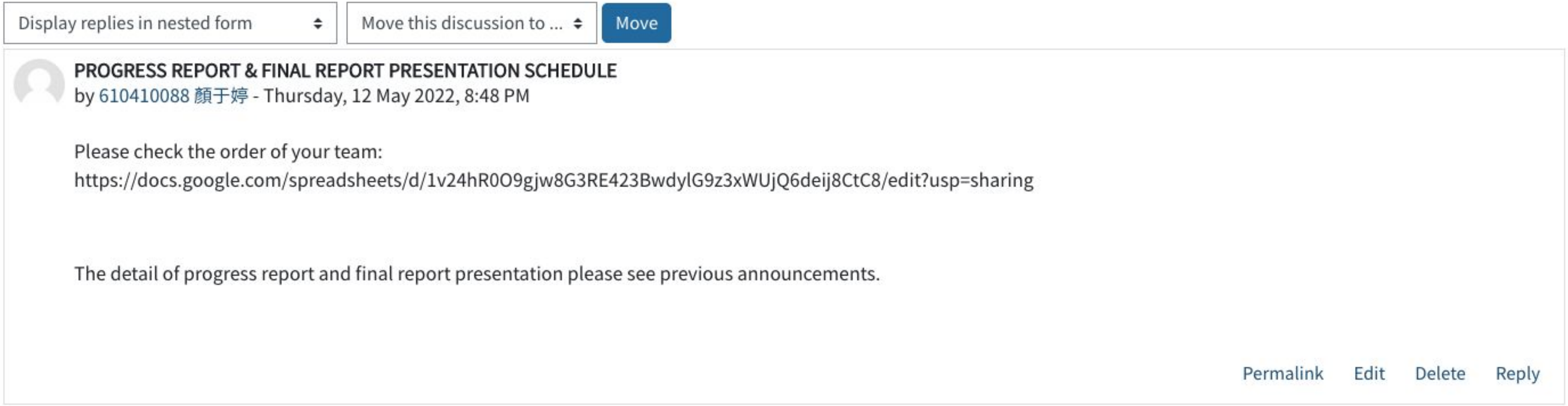
To check if each team is on the right track, they will **present their progress on 5/24 or 5/26 in English or Chinese during the lesson.** Please check the figure below or the syllabus to see the revised schedule.

Each team will need to report their progress in **5 mins** and use 3 to 5 slides to include the following contents:

- introduction of the database system
- current implementation progress of the basic and advanced parts
- list of work distribution (e.g., who did which part)

We will announce each team's order of progress presentation soon.

Every team must present their progress. Otherwise, the team will lose the grade of the final project.



PROGRESS REPORT & FINAL REPORT PRESENTATION SCHEDULE

by 610410088 顏于婷 - Thursday, 12 May 2022, 8:48 PM

Please check the order of your team:
https://docs.google.com/spreadsheets/d/1v24hR0O9gjw8G3RE423BwdylG9z3xWUjQ6deij8CtC8/edit?usp=sharing

The detail of progress report and final report presentation please see previous announcements.

Outline

- File Operations
- Formatted I/O

File Operations

- **Simplicity** is one of the attractions of input and output redirection.
- **Unfortunately**, redirection is **too limited** for many applications.
  - When a program relies on redirection, it has **no control** over its files; it doesn't even know their names.
  - Redirection doesn't help if the program needs to read from **two** files or write to two files at the same time.

File Operations

- When redirection isn't enough, we'll use the **file operations** that `<stdio.h>` provides.
  - Opening a file
  - Closing a file
  - Deleting a file
  - Renaming a file

Opening a File

- **Opening** a **file** for use **as** a **stream** requires a call of the `fopen` function.
- `filename` is the name of the file to be opened.
  - This argument may **include information** about the file's **location**, such as a drive specifier or path.
- `mode` ("mode string") specifies **what operations** we **intend** to perform on the file.
- `restrict` indicates that `filename` and `mode` should point to strings that **don't share** memory locations.

```
FILE *fopen(const char * restrict filename,
            const char * restrict mode);
```

Opening a File

- In **Windows**, be careful when the file name in a call of `fopen` includes the `\` character.
- The call

  ```
  fopen("c:\project\test1.dat", "r")
  ```

  will fail, because `\t` is treated as a **character escape**.
- One way to avoid the problem is to use `\\` instead of `\`:

  ```
  fopen("c:\\project\\test1.dat", "r")
  ```

- An alternative is to use the `/` character instead of `\`:

  ```
  fopen("c:/project/test1.dat", "r")
  ```

Opening a File

- `fopen` returns a file pointer that the program can (and usually will) save in a variable:

  ```
  fp = fopen("in.dat", "r");
    /* opens in.dat for reading */
  ```

- When it can’t open a file, `fopen` returns a null pointer.

```
FILE *fopen(const char * restrict filename,
            const char * restrict mode);
```

Modes

- Factors that determine **which mode** string to pass to `fopen`:
  - **Which operations** are to be performed on the file
  - Whether the file contains **text** or **binary** data

```
// FILE *fopen(const char * restrict filename,
//             const char * restrict mode);

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

Modes

- Mode strings for **text** files:

String	***Meaning***
`"r"`	Open for reading
`"w"`	Open for writing (file need not exist)
`"a"`	Open for **appending** (file need not exist)

?

Modes

- Note that there are **different** mode strings for *writing* data and *appending* data.
- When data is written to a file, it normally **overwrites** what was previously there.
- When a file is opened for **appending**, data written to the file is **added** at the end.

Modes

- Mode strings for text files:

String	***Meaning***
`"r"`	Open for reading
`"w"`	Open for writing (file need not exist)
`"a"`	Open for appending (file need not exist)
`"r+"`	Open for reading and writing, starting at beginning
`"w+"`	Open for reading and writing (truncate if file exists)
`"a+"`	Open for reading and writing (append if file exists)

removing the file contents without deleting the file

Closing a File

- The fclose function allows a program to **close** a file that it's no longer using.
- The argument to fclose must be a **file pointer** obtained from a call of fopen or freopen.
- fclose returns zero if the file was closed successfully.
- Otherwise, it returns the **error code** EOF (a macro defined in <stdio.h>).

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

Closing a File

- The example program that opens a file for reading:

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

#define FILE_NAME "example.dat"

int main(){
     FILE *fp;

     fp = fopen(FILE_NAME, "r");
     if (fp == NULL) {
       printf("Can't open %s\n", FILE_NAME);
       exit(EXIT_FAILURE);
     }
     …
     fclose(fp);
     return 0;
}
```

Closing a File

- It’s also common to see the call of `fopen` **combined** with the declaration of `fp`:

  ```
  FILE *fp = fopen(FILE_NAME, "r");
  ```

  or the test against `NULL`:

  ```
  if ((fp = fopen(FILE_NAME, "r")) == NULL) …
  ```

Attaching a File to an Open Stream

- `freopen` **attaches** a different **file** to a **stream** that’s **already** open.
- The most common use of `freopen` is to **associate** a **file** with one of the **standard streams** (`stdin`, `stdout`, or `stderr`).

```
FILE *freopen(const char * restrict filename,
              const char * restrict mode,
              FILE * restrict stream);
```

Attaching a File to an Open Stream

- A call of `freopen` that causes a program to begin writing to the file `foo`

```
if (freopen("foo", "w", stdout) == NULL) {
     /* error; foo can't be opened */
}
```

Attaching a File to an Open Stream

- A call of `freopen` that causes a program to begin writing to the file `foo`
- If it **can't** open the new file, `freopen` returns a null pointer.

```
FILE *freopen(const char * restrict filename,
             const char * restrict mode,
              FILE * restrict stream);
```

File Operations

- When redirection isn't enough, we'll use the file operations that `<stdio.h>` provides.
  - Opening a file
  - Closing a file
  - **Deleting a file**
  - **Renaming a file**

Remove and Rename Files

- The `remove` and `rename` functions allow a program to perform basic file **management** operations.
- Unlike most other functions in this section, `remove` and `rename` work with file *names* instead of file *pointers*.
- Both functions return zero if they succeed and a nonzero value if they fail.

```
int remove(const char *filename);
int rename(const char *old, const char *new);
```

Remove and Rename Files

- `remove` deletes a file:

```
remove("foo");
  /* deletes the file named "foo" */
```

- The effect of removing a file that’s **currently open** is **implementation-defined.**

Remove and Rename Files

- `rename` changes the name of a file:

```
rename("foo", "bar");
  /* renames "foo" to "bar" */
```

- If a file with the **new name already exists**, the effect is **implementation-defined**.
- `rename` may **fail** if asked to rename an **open file**.

Formatted I/O

- The next group of library functions use **format strings** to control reading and writing.
- These functions include our old friends: `printf` and `scanf`

The …`printf` Functions

- The `fprintf` and `printf` functions write a variable number of data items to an **output stream**, using a **format string** to control the appearance of the output.
- The prototypes for both functions end with the `...` symbol (an ***ellipsis***), which indicates a variable number of additional arguments
- return number of characters written; return negative value when error

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

int printf(FILE * restrict stream, …);
```

The …`printf` Functions

- `printf` always writes to `stdout`, whereas `fprintf` writes to the stream indicated by its first argument
- A call of `printf` is equivalent to a call of `fprintf` with `stdout` as the first argument.

```
printf("Total: %d\n", total); /* writes to stdout */

fprintf(fp, "Total: %d\n", total); /* writes to fp */
```

The …`printf` Functions

- `fprintf` works with any output stream.
- One of its most common uses is to write error messages to `stderr`

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

…`printf` Conversion Specifications

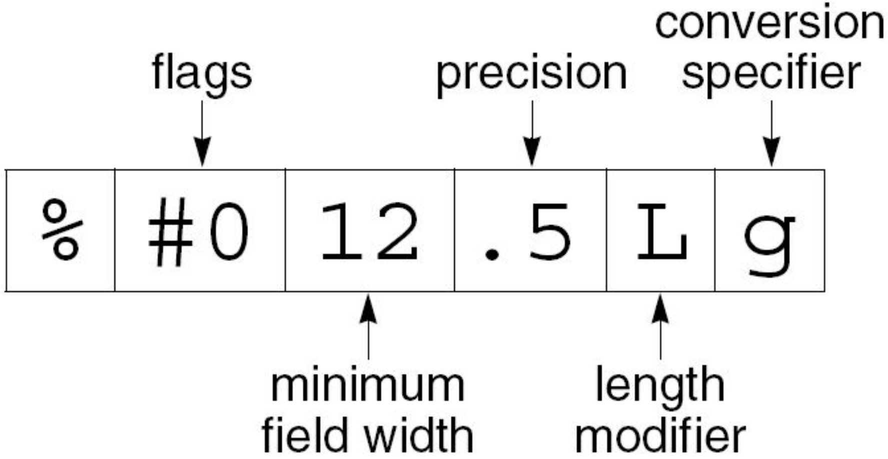
- Both `printf` and `fprintf` require a format string containing ordinary characters and/or **conversion specifications**.
- We introduced them briefly in previous lessons.
- Now, we will add more details.

```
printf("Total: %d\n", total); /* writes to stdout */

fprintf(fp, "Total: %d\n", total); /* writes to fp */
```

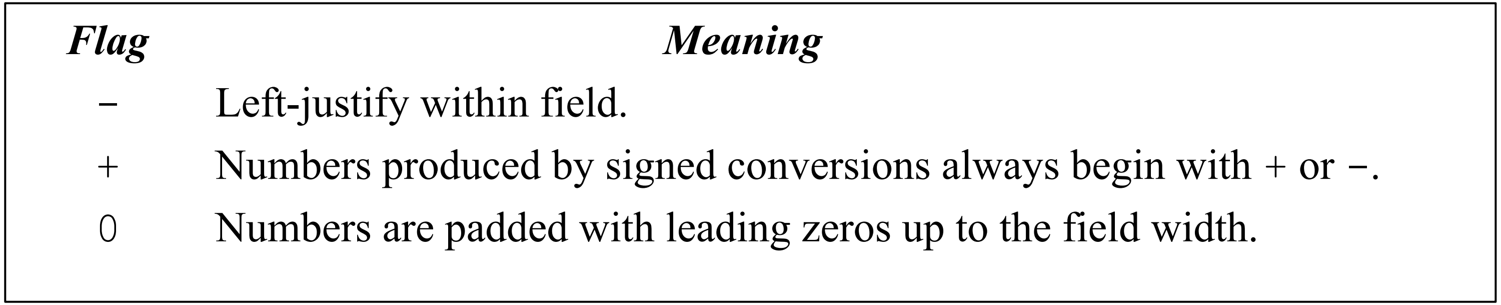
…`printf` Conversion Specifications

- A …`printf` conversion specification consists of the `%` character, followed by as many as five distinct items



Flags (optional; more than one permitted)

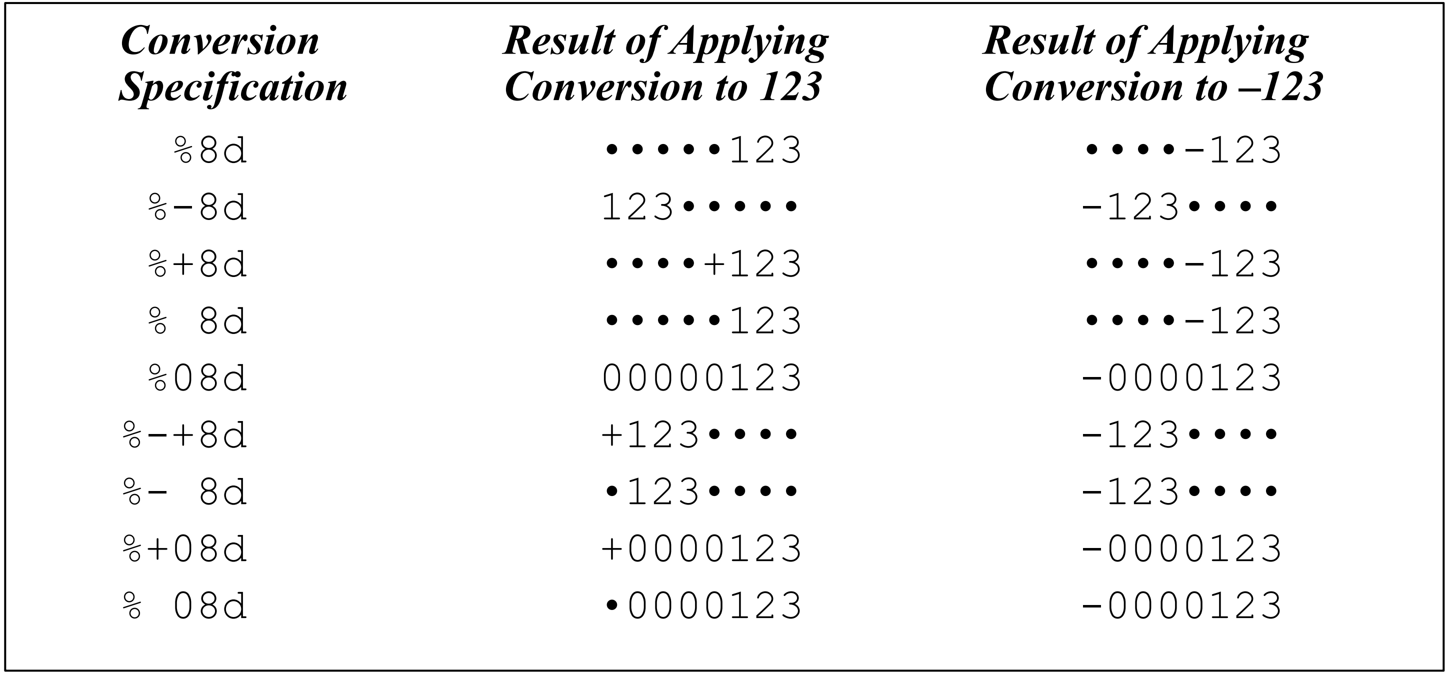
- The - flag causes left justificatrion within a field
- The other flags affect the way numbers are displayed.
- Here are some flag (see Table 22.4 in the textbook for complete table of falgs)



Flag	*Meaning*
−	Left-justify within field.
+	Numbers produced by signed conversions always begin with + or −.
0	Numbers are padded with leading zeros up to the field width.

Flags (optional; more than one permitted)

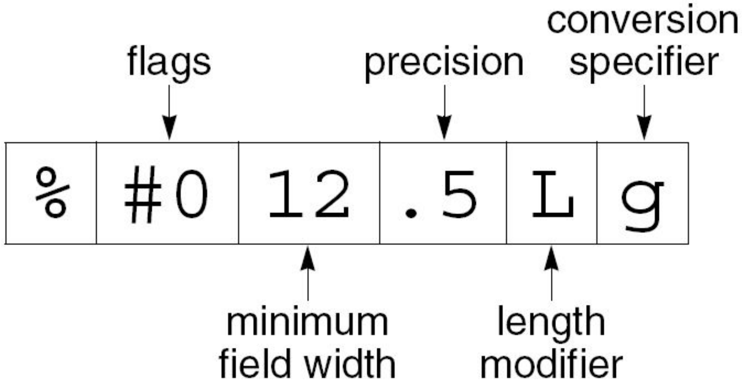
- Examples showing the effect of flags on the `%d` conversion (• represent space)



Conversion Specification	*Result of Applying Conversion to 123*	*Result of Applying Conversion to –123*
`%8d`	`•••••123`	`••••-123`
`%-8d`	`123•••••`	`-123••••`
`%+8d`	`••••+123`	`••••-123`
`% 8d`	`•••••123`	`••••-123`
`%08d`	`00000123`	`-0000123`
`%-+8d`	`+123••••`	`-123••••`
`%- 8d`	`•123••••`	`-123••••`
`%+08d`	`+0000123`	`-0000123`
`% 08d`	`•0000123`	`-0000123`

Minimum field width and Precision (optional).

- We explained these two before, so let’s just quick recap their meaning here.
- Minimum field width:
  - An item that’s too small to occupy the field will be padded.
  - An item that’s too large for the field width will still be displayed in its entirety.
- Precision
  - The meaning of the precision depends on the conversion



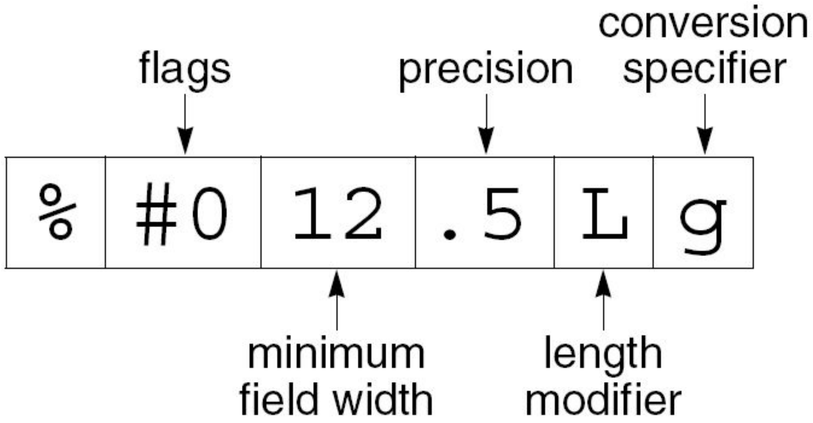
Length modifier (optional)

- Indicates that the item to be displayed has a type that’s longer or shorter than normal.
  - `%d` normally refers to an `int` value; `%hd` is used to display a `short int`
  - `%ld` is used to display a `long int`.
- Here are some Length modifier (see Table 22.5 for complete table of falgs)

Length Modifier	***Conversion Specifiers***	***Meaning***
`h`	`d`, `i`, `o`, `u`, `x`, `X`	`short int, unsigned short int`
`l`	`d`, `i`, `o`, `u`, `x`, `X`	`long int`, `unsigned long int`

Conversion Specifier

- We used the conversion specifiers like `d, c, f, s` the most in prior lessons
- There are more conversion specifiers! Please check the table 22.6.
- Among them, let’s introduce a new useful conversion specifier: `g`



conversion specifier

flags

precision

% #0 12 .5 L g

minimum field width

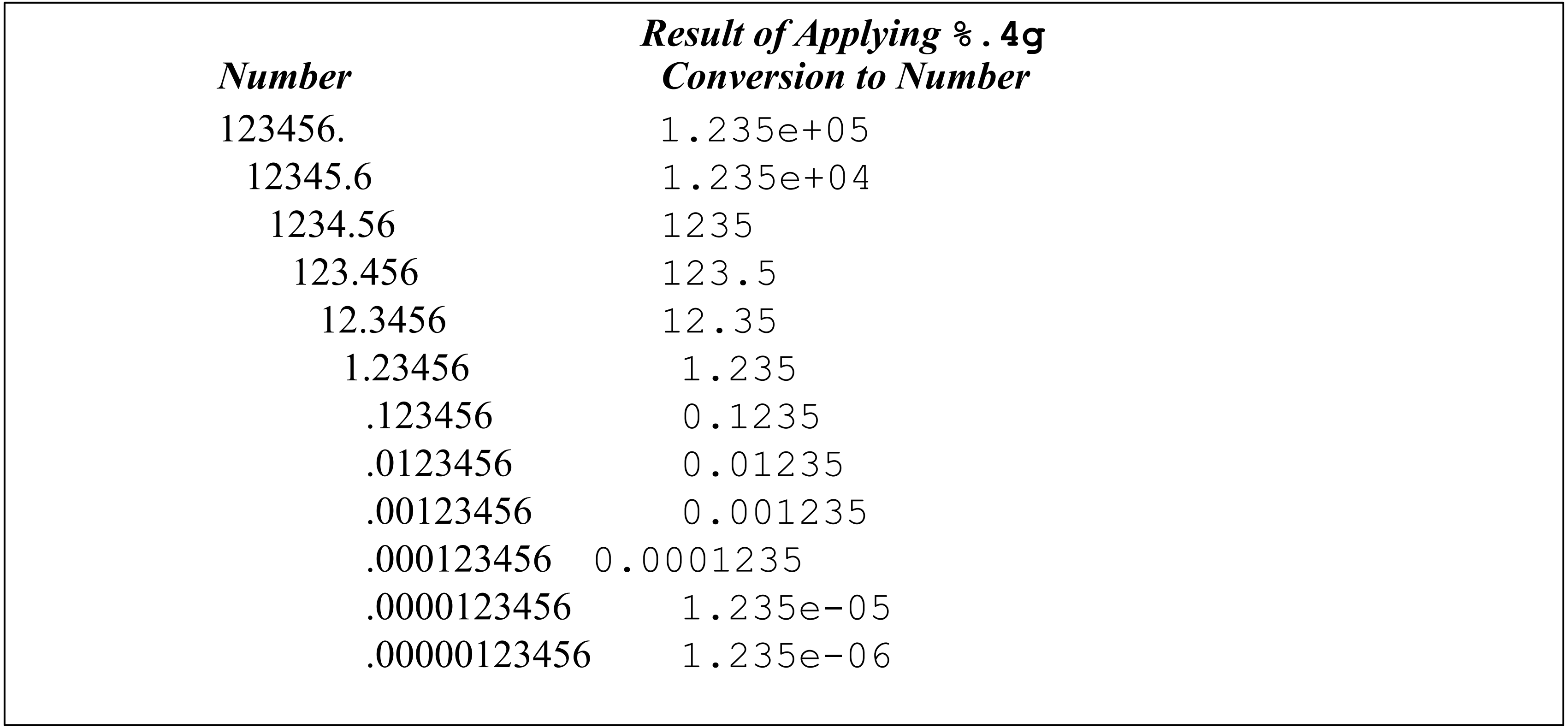
length modifier

Conversion Specifier

- `g:` converts a `double` value to either `f` form or `e` form
  - `e` form is selected if the number's exponet is less than -4
  - or >= to the precision
- Let's check the following example showing how the `%g` conversion displays some numbers in `%e` form and others in `%f` form

Conversion Specifier

precision of `%.4g` is 4



Number	*Result of Applying* **`%.4g`** *Conversion to Number*
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`

* character

- Putting the * character where either number would normally go allows us to specify it as an argument *after* the format string.
- Calls of printf that produce the same output:

```
int i = 10;

printf("%6.4d", i);
printf("%*.4d", 6, i);
printf("%6.*d", 4, i);
printf("%*.*d", 6, 4, i);
```

* character

- A major advantage of * is that it allows us to use a macro to specify the width or precision:

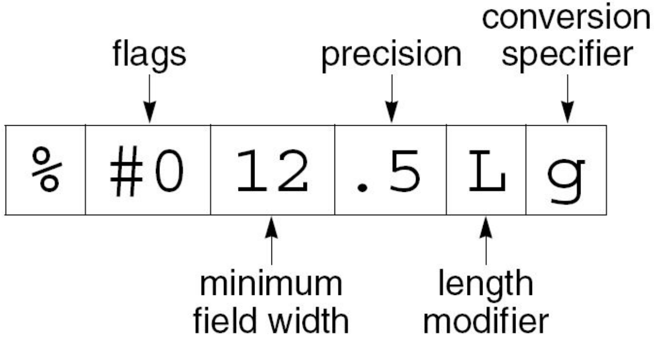
```
printf("%*d", WIDTH, i);
```

- The width or precision can even be computed during program execution:

```
printf("%*d", page_width / num_cols, i);
```

Summary

- File Operations
    - Opening a file
    - Closing a file
    - Deleting a file
    - Renaming a file
- Formatted I/O
    - The `fprintf` and `printf` functions
    - …`printf` conversion specification


flags
precision
conversion specifier
% #0 12 .5 L g
minimum field width
length modifier