Input/Output (3)

Program Design (II)

2022 Spring

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

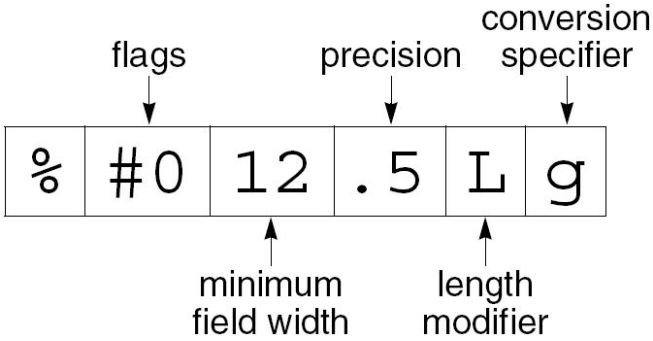
About Final Exam

- 19:10 - 22:00, 2022/6/9 (Thur.)
- It will be an **online** test!
- We will explain how to take it in the **next** lesson.

Recap

- 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

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



Outline

- Formatted I/O: The ...`scanf` Functions
- Character I/O
- Line I/O
- Block I/O

The …`scanf` Functions

- `fscanf` and `scanf` read **data** items from an **input stream**, using a **format string** to indicate the **layout** of the input.
- After the format string, **any** number of **pointers**—each pointing to an **object**—follow as **additional arguments**.
  - the `...` symbol (an ***ellipsis***) indicates a variable number of additional arguments

```
int fscanf(FILE * restrict stream, const char * restrict format, …)
int scanf(const char * restrict format, …)
```

The …`scanf` Functions

- **Input** items are **converted** (according to **conversion specifications** in the format string) and **stored** in these objects.
- `scanf` **always** reads from `stdin`, whereas `fscanf` reads from the **stream** indicated by its first argument
- A call of `scanf` is **equivalent** to a call of `fscanf` with `stdin` as the first argument.

```
int i, j;
FILE *fp = fopen(FILE_NAME, "r");
scanf("%d%d", &i, &j); /* reads from stdin */
fscanf(fp, "%d%d", &i, &j); /* reads from fp */
```

The …`scanf` Functions

- Errors that cause the …`scanf` functions to **return without reading all** input data:
  - ***Input failure***
    - **no more** input characters could be read
  - ***Matching failure***
    - the input characters **didn't match** the format string

The …`scanf` Functions

- The …`scanf` functions **return** the **number** of data **items** that were read and assigned to objects.
- They return `EOF` if an input **failure** occurs **before any data items** can be read.

```
int i, j, k;
k = scanf("%d%d", &i, &j); // k is 2 if read successfully
```

The …`scanf` Functions

- **Loops** that **test** `scanf`'s return value are common.
- A loop that reads a **series** of **integers** one by one, stopping at the first sign of trouble

```
while (scanf("%d", &i) == 1) {
     …
}
```

…`scanf` Format Strings

- Calls of the …`scanf` functions **resemble** those of the …`printf` functions.
- However, there are some **differences**.
- The format string represents a pattern that a …`scanf` function attempts to **match** as it reads input.
  - If the input **doesn't match** the format string, the function **returns**.
  - The input character that didn't match is "**pushed back**" to be **read** in the **future**.

…`scanf` Format Strings

- A …`scanf` format string may contain **three** things:
  - Conversion specifications
  - White-space characters
  - Non-white-space characters

```
int i, j;
scanf("%d, %d", &i, &j);
```

slido

Which conversion specification is showed in this example?

ⓘ Start presenting to display the poll results on this slide.

...`scanf` Format Strings

- ***Conversion specifications.***
  - **Similar** with those in a ...`printf` format string.
  - Most conversion specifications **skip white-space characters** at the **beginning** of an **input item**.
  - Conversion specifications **never skip *trailing*** white-space characters
    - any spaces or tabs **after** the **last non-whitespace** character on the line **until** the newline

```
int r, k;
scanf("%d %d", &r, &k);
scanf("   %d  %d", &r, &k); // same with the above one
scanf("%d %d ", &r, &k); //different from the above one
```

…`scanf` Format Strings

- ***White-space characters.***
  - **One** or **more** white-space characters in a format string match **zero** or **more** white-space characters in the input stream.
- ***Non-white-space characters.***
  - A non-white-space character **other than** `%` matches the same character in the input stream.

```
int i, j;
scanf("%d, %d", &i, &j);
```

…`scanf` Format Strings

The format string `"ISBN %d-%d-%ld-%d"` **specifies** that the input will consist of:

- the letters ISBN
- possibly some white-space characters

…`scanf` Format Strings

The format string `"ISBN %d-%d-%ld-%d"` specifies that the input will consist of:

- an integer
- the – character
- an integer (possibly **preceded** by **white-space** characters)
- the – character
- a long integer (possibly preceded by white-space characters)
- the – character
- an integer (possibly preceded by white-space characters)

…`scanf` Conversion Specifications

- A …`scanf` **conversion specification** consists of the character `%` followed by:
  - `*` (assignment suppression; will not be introduced today)
  - **Maximum field width**
  - **Length modifier**
  - **Conversion specifier**

…`scanf` Conversion Specifications

- ***Maximum field width*** (optional).
  - **Limits** the **number** of **characters** in an input item.
  - **White**-space characters **skipped** at the **beginning** of a conversion don't count.

```
int r;
scanf("%2d", &r);
printf("%d\n", r);
```

```
1234
12
```

…`scanf` Conversion Specifications

- ***Length modifier*** (optional).
  - Indicates that the object in which the input item will be **stored** has a **type** that's **longer** or **shorter** than normal.
- Check table 22.11 for the complete list of length modifiers

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

…`scanf` Conversion Specifications

- ***Conversion specifier***
  - very similar with those for `printf`
  - Check Table 22.12 for the complete list

Conversion Specifier	***Meaning***
`d`	Matches a decimal integer; the corresponding argument is assumed to have type `int *`.
`x`, `X`	Matches a hexadecimal integer; the corresponding argument is assumed to have type unsigned `int *`.
`e`, `E`, `f` , `g`, `G`	Matches a floating-point number; the corresponding argument is assumed to have type `float *`.

slido

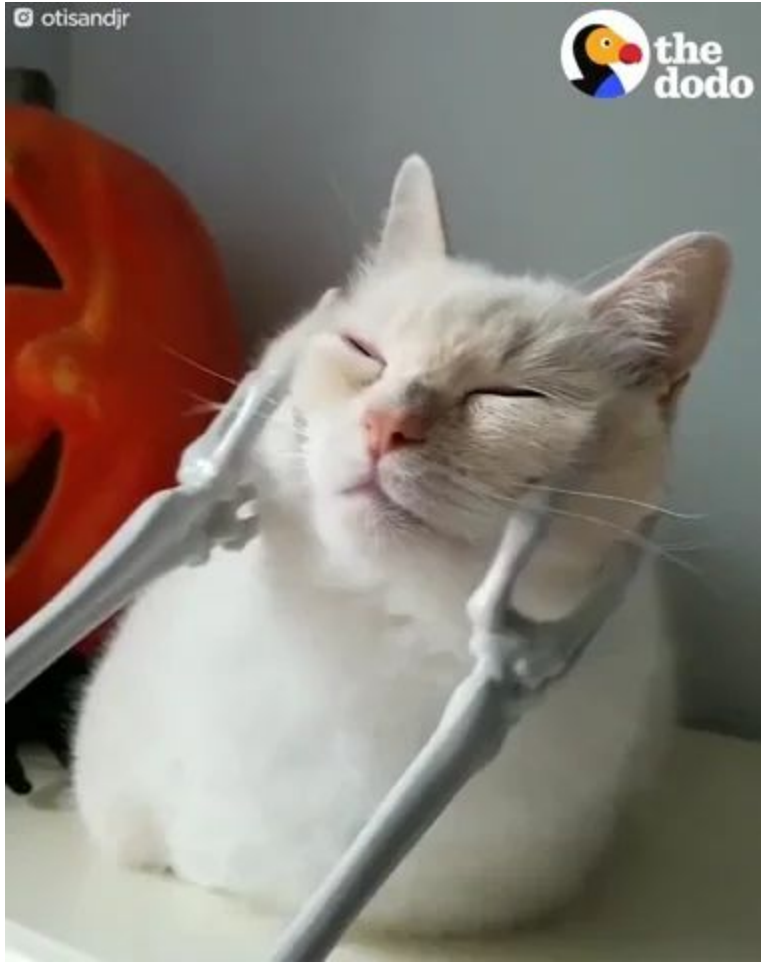
Which one is the correct output if the input is: 12 , 34

ⓘ Start presenting to display the poll results on this slide.

slido

Which one is the correct output if the input is: 12 , 34

ⓘ Start presenting to display the poll results on this slide.


otisandjr
the dodo

Character I/O

- Can read and write **single** characters.
- These functions work equally well with text streams 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.

Character I/O - Output Functions

```
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

- `putchar` writes **one** character to the `stdout` stream:

```
putchar(ch);     /* writes ch to stdout */
```

- `fputc` and `putc` write a character to an arbitrary **stream**:

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

Character I/O - Input Functions

```
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
```

- `getchar` reads a character from `stdin`:

```
ch = getchar();
```

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

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

Character I/O - Input Functions

```
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
```

- These three functions **never return a negative** value other than `EOF`.
- At **end-of-file**, they set the stream's end-of-file indicator and return `EOF`.
- If a read **error** occurs, they set the stream's error indicator and return `EOF`.
- To **differentiate** between the two situations, we can call either `feof` or `ferror`.
  - https://www.tutorialspoint.com/c_standard_library/c_function_feof.htm

Line I/O

- Library functions in the next group are able to read and write **lines**.
- These functions are used **mostly** with **text** streams, although it's legal to use them with binary streams as well.

Line I/O - Output Functions

```
int fputs(const char * restrict s, FILE * restrict stream);
int puts(const char *s);
```

- `puts`: The `puts` function writes a **string** of characters to `stdout`:

  ```
  puts("Hi, there!");  /* writes to stdout */
  ```

- After it writes the characters in the string, `puts` always adds a **new-line** character.

Line I/O - Output Functions

```
int fputs(const char * restrict s, FILE * restrict stream);
int puts(const char *s);
```

- `fputs` is a more **general** version of `puts`.
- Its **second** argument **indicates** the **stream** to which the output should be written:

  ```
  fputs("Hi, there!", fp);  /* writes to FILE *fp */
  ```

- Unlike `puts`, the `fputs` function **doesn't write a new-line** character unless one is present in the string.
- Both return `EOF` if a write error occurs; otherwise, they return a **nonnegative** number.

Line I/O - Input Functions

```
char *gets(char *s);
char *fgets(char * restrict s, int n, FILE * restrict stream);
```

- The `gets` function reads a **line** of input from `stdin`:

  ```
  gets(str); /* reads a line from stdin */
  ```

- `gets` reads **characters one by one**, storing them in the **array** pointed to by `str`, **until** it reads a **new-line** character (which it discards).

Line I/O - Input Functions

```
char *gets(char *s);
char *fgets(char * restrict s, int n, FILE * restrict stream);
```

- `fgets` is a more **general** version of `gets` that can read from any stream.
- `fgets` is also **safer** than `gets`, since it **limits** the **number** of characters that it will store.
- A call of `fgets` that reads a line into a character array named `str`:

  ```
  fgets(str, sizeof(str), fp);
  ```

- `fgets` will read characters **until** it reaches the first new-line character or `sizeof(str)` – 1 characters have been read.

Line I/O - Input Functions

```
char *gets(char *s);
char *fgets(char * restrict s, int n, FILE * restrict stream);
```

- Both `gets` and `fgets` **return a null pointer** if a read **error** occurs or they reach the end of the input stream **before storing any** characters.
- Otherwise, both **return** their **first argument**, which points to the array in which the input was stored.
- Both functions **store a null character** at the end of the string.

Block I/O

- The `fread` and `fwrite` functions 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 as well.

Block I/O - Output Function

```
size_t fwrite(const void * restrict ptr,
              size_t size, size_t nmemb,
              FILE * restrict stream);
```

- `fwrite` is designed to **copy an array from memory to a stream.**
- Arguments in a call of `fwrite`:
  - Address of array (`void * restrict ptr`)
  - Size of each array element (in **bytes**) (`size_t size`)
  - Number of elements to write (`size_t nmemb`)
  - File pointer (`FILE * restrict stream`)

Block I/O - Output Function

```
size_t fwrite(const void * restrict ptr,
              size_t size, size_t nmemb,
              FILE * restrict stream);
```

- A call of `fwrite` that writes the **entire contents** of the array `a` to `fp`

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

- `a`: **name** of **array** is **pointer** to the array
- `sizeof(a[0])`: size of each array element (in **bytes**)
- `sizeof(a) / sizeof(a[0])`: number of elements to write
- `fp`: file pointer

Block I/O - Output Function

```
size_t fwrite(const void * restrict ptr,
              size_t size, size_t nmemb,
              FILE * restrict stream);
```

- `fwrite` **returns** the number of elements **actually written**.
- This number will be **less than the third argument** if a write **error** occurs.

Block I/O - Input Function

```
size_t fread(void * restrict ptr,
             size_t size, size_t nmemb,
             FILE * restrict stream);
```

- `fread` will **read** the elements of an array **from a stream**.
- A call of `fread` that reads the contents of a file into the array `a`:

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

- `fread`'s **return** value indicates the **actual number of elements read**.
- This number should equal the third argument unless the end of the input file was reached or a read error occurred.

Block I/O

- The data **doesn't need** to be in **array** form.
- A call of `fwrite` that writes a structure variable `s` to a file:

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

Summary

- Formatted I/O: The …`scanf` Functions
  - Maximum field width
  - Length modifier
  - Conversion specifier
- Character I/O
  - Input & Output Functions
- Line I/O
  - Input & Output Functions
- Block I/O
  - Input & Output Functions