C 標準輸出入函數庫
與
作業系統

開啟檔案、讀寫檔案

fopen()

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

2. int main(int argc, char **argv)
3. {
4.     FILE* file;
5.     file = fopen("./tmp", "w");
6.     fprintf(file, "this is a tmp file\n");
7.     return 0;
8. }
```

fopen()

- `#include <stdio.h>`
- `FILE *fopen(const char *path, const char *mode);`
- `FILE *fdopen(int fd, const char *mode);`

- 回傳值是FILE這個資料型別
- 初始化FILE只能用stdio定義的函數操作，如：fopen、fdopen
- fdopen可以將上一個章節教的file descriptor轉換為FILE物件

fopen的mode參數

- fopen可以接多種參數，常用的參數如下，修飾參數接在參數之後

參數	代表意義
r	開啟檔案以供讀取（檔案要原本就存在）
w	開啟檔案以供寫入。如果原本就有這個檔案，原先檔案的內容會被清除。原本沒有這個檔案，系統自動建立此檔案。
a	開啟檔案以供寫入，所寫入的資料附加於原檔案之後。原本沒有這個檔案，系統自動建立此檔案。
r+	和「r」一樣，但打開的檔案可供「讀、寫」
w+	和「w」一樣，但打開的檔案可供「讀、寫」
a+	和「a」一樣，但打開的檔案可供「讀、寫」

某些作業系統還提供b這個mode（例如：rb），代表binary，但UNIX並不特別區分「字」與「二進位碼」，因此我們可以忽略b這個mode（如果加上「b」UNIX，也會忽略這個mode）。

Glibc對fopen的擴充

Glibc notes

The GNU C library allows the following extensions for the string specified in mode:

c (since glibc 2.3.3)
Do not make the open operation, or subsequent read and write operations, thread cancellation points. This flag is ignored for **fdopen**().

e (since glibc 2.7)
Open the file with the **O_CLOEXEC** flag. See **open**(2) for more information. This flag is ignored for **fdopen**().

m (since glibc 2.3)
Attempt to access the file using **mmap**(2), rather than I/O system calls (**read**(2), **write**(2)). Currently, use of **mmap**(2) is attempted only for a file opened for reading.

x Open the file exclusively (like the **O_EXCL** flag of **open**(2)). If the file already exists, **fopen**() fails, and sets errno to **EEXIST**. This flag is ignored for **fdopen**().

stdin、stdout、stderr

- 在UNIX內，一啟動程式作業系統就會自動幫我們開啟三個「檔案」，分別是標準輸入、標準輸出及標準錯誤輸出，這三個檔案對應的FILE物件如下

	FILE物件	「通常」的設備
標準輸入	stdin	鍵盤
標準輸出	stdout	螢幕
標準錯誤輸出	stderr	螢幕

自學：fileno

The function fileno() examines the argument stream and returns its integer descriptor.

fprintf

```
#include <stdio.h>
int printf(const char * restrict format, ...)
int fprintf(FILE * restrict stream, const char * restrict format, ...)
int sprintf(char* restrict str, const char * restrict format, ...)
```

- printf將「格式化」後的字串印到標準輸出（通常是螢幕）
- fprintf將「格式化」後的字串印到檔案
- sprintf將「格式化」後的字串印到「記憶體」

Example: fprintf & mode

```
1. #include <stdio.h>
2. int main(int argc, char **argv) {
3.   FILE* file;
4.   file = fopen("./tmp", argv[1]);
5.   fprintf(file, "this_is_a_tmp_file\n");;
6.   fclose(file);
7.   return 0;
8. }
```

執行結果

```
$./write+read2 a+
$./write+read2 a+
$./write+read2 a+
$less ./tmp
this_is_a_tmp_file
this_is_a_tmp_file
this_is_a_tmp_file
(END)
```

- 使用a+打開檔案，讓寫入的資料都附加在檔案之後
- 使用完檔案後，用fclose關閉檔案

檔案位置（position）

- `int fseek(FILE *stream, long offset, int whence);`
- `long ftell(FILE *stream);`

- 這二個函數分別會設定（fseek）和回傳檔案位置（position），特別要注意的是fseek並不會回傳目前的檔案位置，因此要得到檔案位置必須使用ftell
- 和lseek很像，fseek提供三個選項，分別是SEEK_SET, SEEK_CUR, SEEK_END
- 成功回傳0，失敗回傳-1

append + fseek

```
1.  #include <stdio.h>
2.  int main() {
3.      FILE *stream;
4.      char buf[100];
5.      int ret;
6.      stream = fopen("./tmp", "a+");
7.      ret = fseek(stream, 2, SEEK_SET);
8.      fread(buf, 100, 1, stream);
9.      printf(buf);
10.     printf("\nreturn value = %d\n", ret);
11.     fprintf(stream, "append?");
12.     printf("position = %d\n", ftell(stream));
13.     return 0;
14. }
```

結果

```
$./append+fseek
this_is_a_tmp_file
this_is_a_tmp_file
this_is_a_tmp_file
append?
return value = 0
position = 85
$less ./tmp
this_is_a_tmp_file
this_is_a_tmp_file
this_is_a_tmp_file
this_is_a_tmp_file
append?
```

- 使用a, a+打開的stream也可以使用fseek，回傳值為0
- fseek可以改變「讀取位置」，fseek會改變讀取的資料的位置
- 但實際上「寫入的資料會放在檔案的最後面」

C函數庫的buffer

- `#include <stdio.h>`
- `int fflush(FILE *stream);`

- 系統內部有二個重要的buffer，分別位於C函數庫（如果我們使用的是stdio相關的函數），另一個位於核心（Linux kernel），當執行fflush時會將C函數庫內所有被buffer的資料寫到OS。
- 如果想要確保OS的buffer資料也寫入到硬碟則需要使用fsync，但我們只有FILE物件沒有file descriptor。此時可以使用 `int fileno(FILE *stream)` 。

設定buffer的大小

- `#include <stdio.h>`
- `void setbuf(FILE *stream, char *buf);`
- `void setbuffer(FILE *stream, char *buf, size_t size);`
- `void setlinebuf(FILE *stream);`
- `int setvbuf(FILE *stream, char *buf, int mode, size_t size);`

以setvbuf為例

- `int setvbuf(FILE *stream, char *buf, int mode, size_t size);`
  - 依照mode的指示設定stream，並且以buf為buffer，該buffer的大小為size
  - setvbuf必須在「真正使用」stream前使用才會有效果
  - 請記得，使用setvbuf前應該要有這樣的動作：buf=malloc(size);
- mode有三種選項
  - `_IONBF` unbuffered，所有寫入到stream的物件立即寫入到stream
  - `_IOLBF` line buffered，當遇到換行符號（\n）才將該「字串」寫到stream
  - `_IOFBF` fully buffered，當buffer滿了，才將這些物件寫入stream

setvbuf

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

int main(int argc, char **argv)
{
    FILE* stream;
    int bufSize;
    int dataSize = 10000000;
    char *buf;
    int i;

    stream = fopen("./tmp", "w+");
    sscanf(argv[1], "%d", &bufSize);
    buf = (char*)malloc(bufSize);
    setvbuf(stream, buf, _IOFBF, bufSize);
    for (i=0; i<dataSize; i++)
        fwrite("d", 1, 1, stream);
    return 0;
}
```

結果

```
shiwulo@Lonux:~/sp/ch05$ time ./setvbuf 100


real 0m0.449s
user 0m0.164s
sys 0m0.284s
shiwulo@Lonux:~/sp/ch05$ time ./setvbuf 1000


real 0m0.178s
user 0m0.149s
sys 0m0.027s
```

結果

bufsize	real	user	sys
100	10.666s	0.104s	6.248s
1000	0.725s	0.308s	0.168s
10000	0.239s	0.168s	0.016s

讀寫檔案

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

- 以fread為例，從stream讀取nmemb筆資料，每筆資料的大小為size這麼大，到ptr所指向的buffer
- 以fwrite為例， 從ptr所指向的buffer的資料，寫出到stream。共寫出nmemb筆資料，每筆資料的大小為size這麼大
- fread及fwrite的回傳值都是「讀取幾筆資料」，不是讀取多少個字元

Errors 及 EOF

```
int feof(FILE *stream);
int ferror(FILE *stream);
void clearerr(FILE *stream);
```

- 以fread為例，不管是發生錯誤或者讀到檔案結尾，回傳值都小於預期（例如小於要寫出的資料量）。那我們要怎樣區分EOF和Error?
- feof、ferror可以分別測出到底是檔案結尾或者是錯誤
- 測試完畢以後呼叫clearerr清除該「錯誤標示」

feof

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

2. int main() {
3.     FILE *stream;
4.     char buf[5000];
5.     int ret;
6.     stream = fopen("./tmp", "a+");
7.     ret=fread(buf, 10, 500, stream);
8.     printf("ret = %d\n", ret);
9.     ret=fread(buf, 10, 500, stream);
10.    printf("ret = %d\n", ret);
11.    if(ret!=1) {
12.        if (ferror(stream))
13.            printf("error\n");
14.        if (feof(stream))
15.            printf("EOF\n");
16.    }
17.    return 0;
18.}
```

小結

- 使用stream雖然比較方便，速度也往往比較快，但是要注意glibc如何管理buffer
- 較大的buffer速度比較快，但萬一系統斷電或當機，也要冒比較大的風險

寬字串（wide string, i18n）

strlen.c

```c
#include <stdio.h>
#include <wchar.h>
#include <string.h>

int main(int argc, char **argv)
{
    char *str = "中文";
    printf("%d\n", (int)strlen(str));
    return 0;
}
```

執行結果

```
$ ./strlen
6
```

- 『中文』是二個字，但strlen卻告訴我們『中文』是三個字
- 這是因為中文是unicode編碼非ASCII編碼

wcslen.c

```c
#include <stdio.h>
#include <wchar.h>
#include <string.h>

int main(int argc, char **argv)
{
    wchar_t* wstr = L"中文";
    printf("%d\n", (int)wcslen(wstr));
    return 0;
}
```

執行結果

```
$./wcslen
2
```

- wchar_t* wstr = L“中文”;宣告wstr是寬字元字串
- 使用wcslen（寬字元版本的strlen）就可以正確的判斷出字串的長度

fwide()

```
int fwide (FILE *stream, int mode)
```

- mode為0時，回傳stream目前的讀寫狀態。大於0切換為讀寫寬字串，小於0切換為讀寫一般字串

寬字元的讀取及寫入

```
#include <wchar.h>
wchar_t *fgetws(wchar_t *ws, int n, FILE
*stream);
int fputws(const wchar_t *ws, FILE *stream);
```

- 寬字元（萬國碼，unicode）幾乎都定義在<wchar.h>
- 例如fgetws可以從stream中讀取一個寬字元字串

製造一個系統唯一的暫存檔

tmp file

- `char *tempnam(const char *dir, const char *pfx);`
- `FILE *tmpfile(void);`
- `char *tmpnam(char *s);`
- `char *mktemp(char *template);`
- `int mkstemps(char *template, int suffixlen);`

以mktemp為例

- `char *mktemp(char *template);`
- 在系統中建立一個「唯一的檔案」
- template的格式為「最後6個字母必須是XXXXXX（一定要大寫）」
- XXXXXX會被替換成一個「唯一的字串」，確保這個檔案的檔名在系統中是唯一
- 通常用來製造暫存檔案

mktemp

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <errno.h>

4.  int main() {
5.      FILE* stream;
6.      char tmpStr[] = "./shiwulo_XXXXXX";
7.      mktemp(tmpStr);
8.      printf("%s\n", tmpStr);
9.      stream = fopen(tmpStr, “w+”);   /*權限為600，只有owner才可以讀寫*/
10.     if (stream == NULL)
11.         perror("error: ");
12.     fputs("hello\0", stdout);
13.     return 0;
14. }
```

俺的建議

- 使用FILE *tmpfile(void)比較好，因為製造tempName後，在還沒打開這個檔案時，有可能另外一隻程式剛好使用同樣的tempName做同樣的事情
  - 這種情況很少見
  - 萬一發生這種情況，bug很難抓
- FILE *tmpfile(void)可以一次搞定製造檔名和開檔案

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

NAME
       mkstemp, mkostemp, mkstemps, mkostemps - create a unique temporary file

SYNOPSIS
       #include <stdlib.h>

       int mkstemp(char *template);

       int mkostemp(char *template, int flags);

       int mkstemps(char *template, int suffixlen);

       int mkostemps(char *template, int suffixlen, int flags);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       mkstemp():
           _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 500 ||
           _XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED
           || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200112L

       mkostemp(): _GNU_SOURCE
       mkstemps(): _BSD_SOURCE || _SVID_SOURCE
       mkostemps(): _GNU_SOURCE

DESCRIPTION
       The mkstemp() function generates a unique temporary filename from  tem-
       plate,  creates and opens the file, and returns an open file descriptor
       for the file.

       The last six characters of template must  be  "XXXXXX"  and  these  are
       replaced  with  a string that makes the filename unique.  Since it will
       be modified, template must not be a  string  constant,  but  should  be
       declared as a character array.

       The file is created with permissions 0600, that is, read plus write for
       owner only.  The returned file descriptor provides both read and  write
       access  to  the file.  The file is opened with the open(2) O_EXCL flag,
       guaranteeing that the caller is the process that creates the file.

       The mkostemp() function is like mkstemp(), with the difference that the
       following bits—with the same meaning as for open(2)—may be specified in
       flags: O_APPEND, O_CLOEXEC, and O_SYNC.  Note that  when  creating  the
       file, mkostemp() includes the values O_RDWR, O_CREAT, and O_EXCL in the
       flags argument given to open(2); including these values  in  the  flags
       argument  given  to  mkostemp()  is unnecessary, and produces errors on
       some systems.

       The mkstemps() function is like mkstemp(), except that  the  string  in
       template  contains a suffix of suffixlen characters.  Thus, template is
       of the form prefixXXXXXXsuffix, and the string XXXXXX  is  modified  as
       for mkstemp().

       The   mkostemps()  function  is  to  mkstemps()  as  mkostemp()  is  to
       mkstemp().

RETURN VALUE
       On success, these functions return the file descriptor of the temporary
       file.  On error, -1 is returned, and errno is set appropriately.
```

俺的建議

- 使用左邊列出來的這些函數，他們都將1. 產生檔名、開啟檔案，合而為一
- 這些類似的函數，請自學

小結

- 要注意寬字元的函數跟一般的函數的不同
- 了解怎樣在系統內產生temp file

作業

- 寫一隻程式acp，他在複製檔案的時候會先製造一個tmpFile，等到檔案複製結束，再用move的方式，將檔案移動到指定的位置，並給予指定的檔案
- 範例：acp file1 file2，將file1複製到file2，但是複製的過程先產生暫存檔案，等複製結束，再將暫存檔案move到file2的位置