

# Ch15- Standard Input/Output & Socket

Instructor: Limei Peng  
Dept. of CSE, KNU

# Ch15- Standard Input/Output & Socket

0157 CH15

Instructor: Limei Peng  
Dept. of CSE, KNU

# Outlines

- Review on C file operation functions
- Standard input/output operations (on files)
  - <sup>file</sup>**fopen()**
  - **fdopen()**
  - **fileno()**

# System function calls VS. Standard input/output

## ❑ System input/output functions

- read()
- write()
- open()
- close()
- ...

## ❑ Standard input/output functions

- fopen()
- fclose()
- fdopen()
- fileno()
- ...

# Standard I/O

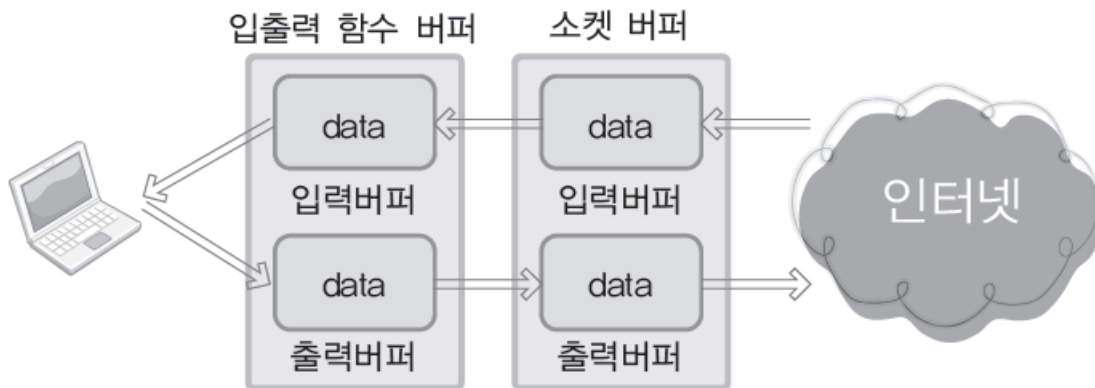
- ❑ Before a file can be read or written, it is opened by the library function **fopen**.
  - **fopen**: an external name like "data.txt"; does housekeeping and negotiation with the operating system
  - **fopen**: returns a **pointer** to be used in the following reading or writing operations of the file
- ❑ The pointer returned by **fopen** points to a structure type called **FILE**, e.g.,  
**FILE** \*fp;

# 표준 입출력 함수의 두 가지 장점

- 표준 입출력 함수는 이식성(Portability)이 좋다.
- 표준 입출력 함수는 버퍼링을 통한 성능의 향상에 도움이 된다.

- Socket buffer: for TCP reliability
- Input/output buffer: for performance improvement, i.e., the more data, the better performance (transmission in a bundle).
- E.g., transmit 1 byte for 10 times v.s. transmit 10 bytes once

ANSI C 기반의 표준 입출력 함수는 모든 컴파일러에서 지원을 하기 때문에 이식성이 좋아진다.



표준 입출력 함수를 이용해서 데이터를 전송할 경우 왼쪽의 그림과 같이 소켓의 입출력 버퍼 이외의 버퍼를 통해서 버퍼링이 된다.

# 표준 입출력 함수의 사용에 있어서 불편사항

- 양방향 통신이 쉽지 않다.
- 상황에 따라서 fflush 함수의 호출이 빈번히 등장할 수 있다.
- 파일 디스크립터를 FILE 구조체의 포인터로 변환해야 한다.

int fflush(File \*stream):

- Causes the system to empty the buffer that is associated with the specified output stream if possible.
- Moves the data from the buffers to the specified physical file.
- Called when changing the status from writing to reading.

fopen 함수 호출 시 반환되는 File 구조체의 포인터를 대상으로 입출력을 진행할 경우, 입력과 출력이 동시에 진행되게 하는 것은 간단하지 않다. 데이터가 버퍼링 되기 때문이다!

소켓 생성시 반환되는 것은 파일 디스크립터이다. 그런데 표준 C 함수에서 요구하는 것은 FILE 구조체의 포인터이다. 따라서 파일 디스크립터를 FILE 구조체의 포인터로 변환해야 한다.

# 표준 입출력 함수와 시스템 함수의 성능비교

```
int main(int argc, char *argv[])
{
    int fd1, fd2;    // fd1, fd2에 저장되는 것은 파일 디스크립터!
    int len;
    char buf[BUF_SIZE];

    fd1=open("news.txt", O_RDONLY);
    fd2=open("cpy.txt", O_WRONLY|O_CREAT|O_TRUNC);

    while((len=read(fd1, buf, sizeof(buf)))>0)
        write(fd2, buf, len);

    close(fd1);
    close(fd2);
    return 0;
}
```

예제 syscopy.c

왼쪽의 경우 시스템 함수를 이용해서 버퍼링 없는 파일 복사를 진행하고 있다.

아래의 경우 표준 입출력 함수를 이용해서 버퍼링 기반의 파일 복사를 진행하고 있다.

```
int main(int argc, char *argv[])
{
    FILE * fp1; // fp1에 저장되는 것은 FILE 구조체의 포인터
    FILE * fp2; // fp2에 저장되는 것도 FILE 구조체의 포인터
    char buf[BUF_SIZE];

    fp1=fopen("news.txt", "r");
    fp2=fopen("cpy.txt", "w");

    while(fgets(buf, BUF_SIZE, fp1)!=NULL)
        fputs(buf, fp2);

    fclose(fp1);
    fclose(fp2);
    return 0;
}
```

예제 stdcopy.c

300메가 바이트 이상의 파일을 대상으로 테스트 시 속도의 차가 매우 극명하게 드러난다.



# fdopen() : FILE 구조체 포인터로의 변환

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

```
FILE * fdopen(int fildes, const char * mode);
```

➔ 성공 시 변환된 FILE 구조체 포인터, 실패 시 NULL 반환

- fildes      변환할 파일 디스크립터를 인자로 전달.
- mode      생성할 FILE 구조체 포인터의 모드(mode)정보 전달.

- **mode**: the most frequently used modes are "w" and "r"  
↳ output                      ↳ input
- **FILE\***: file pointer

# Example#2-fdopen()

```
/*desto.c*/
#include<stdio.h>
#include<fcntl.h>

int main()
{
    FILE* fp;
    int fd=open("test.txt", O_WRONLY|O_CREAT|O_TRUNC);
    if(fd == -1)
    {
        fputs("file open error", stdout);
        return -1;
    }

    fp=fdopen(fd, "w");
    fputs("Networking C Programming\n", fp);
    fclose(fp);
    return 0;
}
```

모든 값이 0.

```
socket@ubuntu:~/Desktop/code$ vi desto.c
socket@ubuntu:~/Desktop/code$ gcc desto.c -o do
socket@ubuntu:~/Desktop/code$ ./do
socket@ubuntu:~/Desktop/code$ cat test.txt
Networking C Programming
socket@ubuntu:~/Desktop/code$
```

## fileno() : 파일 디스크립터로의 변환

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

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

➔ 성공 시 변환된 파일 디스크립터, 실패 시 -1 반환

- ❑ Convert a file pointer, i.e., **FILE\***, to a file descriptor (int)

# Example#3-fdopen() & fileno()

```
/*todes.c*/
#include<stdio.h>
#include<fcntl.h>

int main()
{
    FILE *fp;
    int fd = open("data.dat", O_WRONLY|O_CREAT|O_TRUNC);
    if(fd==-1)
    {
        fputs("file open error", stdout);
        return -1;
    }

    printf("First file descriptor: %d\n", fd);
    fp=fdopen(fd, "w");
    fputs("TCP socket programming \n", fp);
    printf("Second file descriptor: %d\n", fileno(fp));
    fclose(fp);
    return 0;
}
```

*Handwritten notes:*

- Arrows pointing from `fd` in `fdopen` to `fd` in `printf` and `fileno`.
- A box around `"TCP socket programming \n"` with an arrow pointing to `fp` in `fputs`.
- Handwritten text: `output를 위한.` (for output).

```
socket@ubuntu:~$ vi todes.c
socket@ubuntu:~$ gcc todes.c -o td
socket@ubuntu:~$ ./td
First file descriptor: 3
Second file descriptor: 3
socket@ubuntu:~$
```

# Example#4: Socket based on standard I/O (1/4)

(이거 반까지 끝 (연결함의 함수))

```
/* echo_stdserv.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```
#define BUFSIZE 1024
void error_handling(char *message);
```

```
int main(int argc, char **argv)
{
```

```
    int serv_sock;
    int clnt_sock;
    FILE* readFP;
    FILE* writeFP;
    char message[BUFSIZE];
```

```
    struct sockaddr_in serv_addr;
    struct sockaddr_in clnt_addr;
    int clnt_addr_size;
```

```
    if(argc!=2){
        printf("Usage : %s <port>\n", argv[0]);
        exit(1);
    }
```

```
    serv_sock=socket(PF_INET, SOCK_STREAM, 0);
    if(serv_sock == -1)
        error_handling("socket() error");
```

```
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family=AF_INET;
    serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
    serv_addr.sin_port=htons(atoi(argv[1]));
```

```
    if(bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr))== -1)
        error_handling("bind() error");
```

```
    if(listen(serv_sock, 5)== -1)
        error_handling("listen() error");
```

## Server

# Example#4: Socket based on standard I/O (2/4)

```
clnt_addr_size=sizeof(clnt_addr);
clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr,&clnt_addr_size);
if(clnt_sock==-1)
    error_handling("accept() error");
/* convert file descriptor to file pointer*/
readFP=fdopen(clnt_sock, "r");
writeFP=fdopen(clnt_sock, "w");

/* send and receive data */
while(!feof(readFP)){
    fgets(message, BUFSIZE, readFP);
    fputs(message, writeFP);
    fflush(writeFP);
}
fclose(writeFP);
fclose(readFP);

return 0;
}

void error_handling(char *message)
{
    fputs(message, stderr);
    fputc('\n', stderr);
    exit(1);
}
```

Server  
(Cont.)

# Example#4: Socket based on standard I/O (3/4)

```
/*echo_stdclient.c*/  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <arpa/inet.h>  
#include <sys/socket.h>
```

```
#define BUF_SIZE 1024  
void error_handling(char* message);
```

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

```
    int sock;  
    char message[BUF_SIZE];  
    int str_len;  
    struct sockaddr_in serv_addr;
```

```
    FILE* readfp;  
    FILE* writefp;
```

*fps (one for reading,  
another one for writing)*

```
    if( argc != 3 )  
    {  
        printf("Usage : %s<IP> <port>\n", argv[0]);  
        exit(1);  
    }
```

```
    sock = socket(PF_INET, SOCK_STREAM, 0);
```

```
    if( sock == -1 )  
        error_handling("socket() error");
```

```
    memset(&serv_addr, 0, sizeof(serv_addr));  
    serv_addr.sin_family = AF_INET;  
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]);  
    serv_addr.sin_port = htons(atoi(argv[2]));
```

```
    if( connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) == 1 )  
        error_handling("connect() error");
```

```
    else  
        puts("connected.....");
```

## Client



# Example#4: Socket based on standard I/O (4/4)

```
readfp = fdopen(sock, "r");  
writefp = fdopen(sock, "w");
```

```
while(1)  
{  
    fputs("Input message(Q to quit): ", stdout);  
    fgets(message, BUF_SIZE, stdin);  
    if( !strcmp(message, "q\n") || !strcmp(message, "Q\n"))  
        break;  
  
    fputs(message, writefp);  
    fflush(writefp);  
    fgets(message, BUF_SIZE, readfp);  
    printf("Message from server:%s", message);  
}  
  
close(sock);  
return 0;  
}
```

Client(Cont.)

입력용, 출력용 FILE 구조체 포인터를 각각 생성해야 한다.

표준 C 입출력 함수를 사용할 경우 소켓의 버퍼 이외에 버퍼링이 되기 때문에 필요하다면, fflush 함수를 직접 호출해야 한다.

## 일반적인 순서

1. 파일 디스크립터를 FILE 구조체 포인터로 변환
2. 표준 입출력 함수의 호출
3. 함수 호출 후 fflush 함수호출을 통해서 버퍼 비움

```
void error_handling(char* message)  
{  
    fputs(message, stderr);  
    fputc('\n', stderr);  
    exit(1);  
}
```

```
socket@ubuntu:~/Desktop/code$ vi echo_stdclient.c  
socket@ubuntu:~/Desktop/code$ gcc echo_stdclient.c -o esc  
socket@ubuntu:~/Desktop/code$ ./esc 127.0.0.1 9190  
connected.....  
Input message(Q to quit): Testing standard input/output in socket  
Message from server:Testing standard input/output in socket  
Input message(Q to quit): q  
socket@ubuntu:~/Desktop/code$
```



Auxiliary 

# Review on File operations

- ❑ Open a file: `fopen()`
- ❑ Close a file: `fclose()`
- ❑ <sup>input</sup>  
Read a binary file: `fgetc()`, `getc()`, `fgets()`, `gets()`,  
`fscanf()`, `fread()`
- ❑ <sup>output</sup>  
Write a binary file: `fputc()`, `putc()`, `fputs()`, `puts()`,  
`fprintf()`, `fwrite()`














# fopen()

```
FILE *fopen(char *name, char *mode) ;  
void fclose(FILE* stream)
```

- ❑ **fopen** returns a pointer to a **FILE**
- ❑ **mode** can be
  - "r" - read
  - "w" - write
  - "a" - append
  - "b"- binary; can be appended to the mode string to work with binary files. For example, "rb" means reading binary file.

# Create and write to a file

```
#include <stdio.h>
int main()
{
    FILE *fp;
    char name[10];
    double balance;
    int account;
    if ((fp = fopen("clients.dat", "w")) == NULL) {
        printf("File could not be opened\n");
    }
    else {
        printf("Enter one account, name, and balance.\n");
        scanf("%d%s%lf", &account, name, &balance);
        outputfprintf(fp, "%d %s %.2f\n", account, name, balance);
        fclose(fp);
    }
    return 0;
}
```

 bin	2012/9/4 下午 07...	檔案資料夾
 obj	2012/9/4 下午 07...	檔案資料夾
 clients.dat	2012/5/16 下午 0...	DAT 檔案
 fact	2012/3/5 下午 07...	C source file
 hello	2012/3/6 上午 12...	project file
 hello.depend	2012/3/20 下午 1...	DEPEND 檔案
 hello.layout	2012/3/20 下午 1...	LAYOUT 檔案
 main	2012/5/30 上午 0...	C source file
 main	2012/5/30 上午 0...	應用程式
 main.o	2012/5/30 上午 0...	O 檔案
 Makefile	2012/3/20 下午 1...	C source file
 mymath	2012/3/5 下午 07...	Header file
 power	2012/3/5 下午 07...	C source file

clients - 記事本

檔案(F) 編輯(E) 格式(O) 檢視(V) 說明(H)

```
100 paul 100.00
```

# Text Files - read from a file

```
#include <stdio.h>
int main()
{
    FILE *fp;
    char name[10];
    double balance;
    int account;
    if ((fp = fopen("clients.dat", "r")) == NULL) {
        printf("File could not be opened\n");
    }
    else {
        fscanf(fp, "%d%s%lf", &account, name, &balance);
        printf("%d %s %.2f\n", account, name, balance);
        fclose(fp);
    }
    return 0;
}
```

# Text Files - character I/O

```
int fputc(int c, FILE *fp);
```

```
int putc(int c, FILE *fp);
```

- Write a character **c** to a file. **putc()** is often implemented as a **MACRO** (hence faster).
- On success, the character written is returned.
- On error, EOF is returned

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

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

- Read a character **c** to a file. **getc()** is often implemented as a **MACRO** (hence faster).
- On success, the character read is returned.
- On error, EOF is returned

# Text Files - character I/O

---

```
#include <stdio.h>
int main()
{
    FILE *source_fp, *dest_fp;
    int ch;

    if ((source_fp = fopen("source.txt", "r")) == NULL)
        printf("cannot open source file\n");
    if ((dest_fp = fopen("dest.txt", "w")) == NULL)
        printf("cannot open dest file\n");
    while ((ch = getc(source_fp)) != EOF)
        putc(ch, dest_fp);

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



# Text Files-standard input & output

- ❑ `FILE *stdin` // screen input as a file
- ❑ `FILE *stdout` // screen output as a file

# Text Files - stdin, stdout

```
#include <stdio.h>

int main ()
{
    int c;
    while ((c = fgetc(stdin)) != '*')
    {
        fputc(c, stdout);
    }
}
```

# Text Files - Line I/O

int **fputs** (const char \*s, FILE \*fp) ;

- Write a line of characters to a file.
- On success, a non-negative value is returned.
- On error, the function returns EOF

char\* **fgets** (char \*s, int n, File \*fp) ;

- Read characters from a file until it reaches the first new-line or (n-1) characters, in which it places the NULL character ('\0') at the end of the string.
- On success, the function returns s.
- If the end-of-file is encountered before any characters could be read, the pointer returned is a NULL pointer (and the contents of s remain unchanged).

# Text Files - Line I/O

```
#include <stdio.h>
int main()
{
    FILE *source_fp, *dest_fp;
    char s[100];

    if ((source_fp = fopen("source.txt", "r")) == NULL)
        printf("cannot open source file\n");
    if ((dest_fp = fopen("dest.txt", "w")) == NULL)
        printf("cannot open dest file\n");

    while (fgets(s, 100, source_fp) != NULL)
        fputs(s, dest_fp);

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