18. 파일(file)

- **핸들(handle)**을 사용하는 핸들계 파일 조작함수들만을 다룰 것입니다.
- C++표준 라이브러리인 iostream에서 지원하는 파일 조작 클래스(fstream)에 대해서도 언급하지 않을 것입니다.

핸들(handle)이란?

• **핸들(handle)**이란 운영체제(operating system)가 특정한 정보를 유지하기 위해서, 메모리에 유지하는 정보 블록(information block)에 붙여진 고유 번호(unique number)를 말합니다.

• 사용자가 파일에 관해서 입출력 작업을 하기 위해서는 먼저 **파일 핸들(file handle)**을 얻어야 합니다.
• 파일 핸들은 **파일 제어 블록(FCBs: File Control Blocks)**에 붙여진 고유 번호입니다.
• 파일 제어 블록은 디스크에 존재하는 파일에 입출력 작업을 하기 위해서 다양한 정보를 유지하고 있는 블록입니다.
• 파일 제어 블록을 메모리에 할당한 다음, 디스크 파일에 관한 정보로 이 구조체 블록의 필드를 초기화하는 것을 **파일을 연다(open a file)**고 합니다.

open("파일이름","파일모드");

• 파일에 관한 함수는 크게 다음과 같이 구분할 수 있습니다.

① 파일 열기(open),닫기(close) 함수
② 파일 입출력(read/write) 함수
③ 파일 포인터(file pointer) 조작 함수
④ 기타(etc.) 함수

① 파일 열기(open),닫기(close)

• 파일 제어 블록을 할당하고, 핸들을 얻는 것을 '파일을 연다'고 합니다.
• 메모리에 할당된 파일 제어블록을 해제하는 것을 '파일을 닫는다'고 합니다.
• open(), close()가 각각 이 일을 담당합니다.
• open()은 파일 핸들을 리턴하고, close()는 파일 핸들을 파라미터로 받아야 할 것입니다.

② 파일 입출력(read/write)

• 파일 제어 블록의 핸들을 이용해서 디스크 파일로부터 읽고(read()), 쓰는(write())일을 하는 함수들입니다.
• 이러한 함수들의 첫 번째 파라미터는 파일 핸들이어야 할 것입니다.

③ 파일 포인터(file pointer) 조작

• 파일 포인터의 위치를 설정하는 함수들입니다.
• 파일 포인터를 처음으로 혹은 끝으로 옮기거나, 특정한 위치로 이동시킵니다.
• lseek(), seek()등의 함수가 있습니다.
• 입출력 함수들은 파일 포인터를 기준으로 작업합니다. 예를 들면, read()함수는 현재 파일 포인터가 가리키는 곳에서 읽기를 시작합니다.

④ 기타(etc.)

• 파일을 끝인지를 검사하는 eof(), 파일의 길이를 구하는 getlength(), 파일의 크기를 변경시키는 chgsize(), 파일을 제거하는 remove(), 파일 모드를 설정하는 setmode()와 파일을 락(lock)시키는 lock()등이 있습니다.

파일의 사용

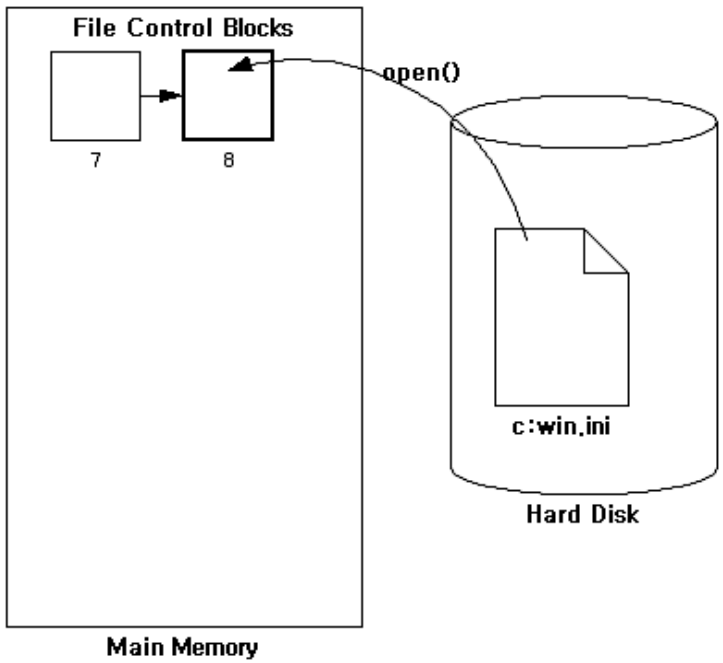
• 제일 먼저 해야 할 일은 open()을 사용하여 파일 제어 블록(FCB)을 할당받고 핸들을 얻는 것입니다.
• 예를 들면, C:₩Windows₩Win.ini 파일을 이진 파일(binary file)로 읽고 쓰기 위해서 열려면 다음 문장을 사용합니다.

```
int handle;
handle=open("c:\\windows\\win.ini",O_RDWR|O_BINARY);
```

• open()의 첫 번째 파라미터는 파일명이며, 두 번째 파라미터는 파일 모드를 비트 플래그로 설정합니다.

```
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>
```

- open()의 리턴 값을 받기 위해 단지 정수(integer)만을 사용했다는 것에 주목하세요.



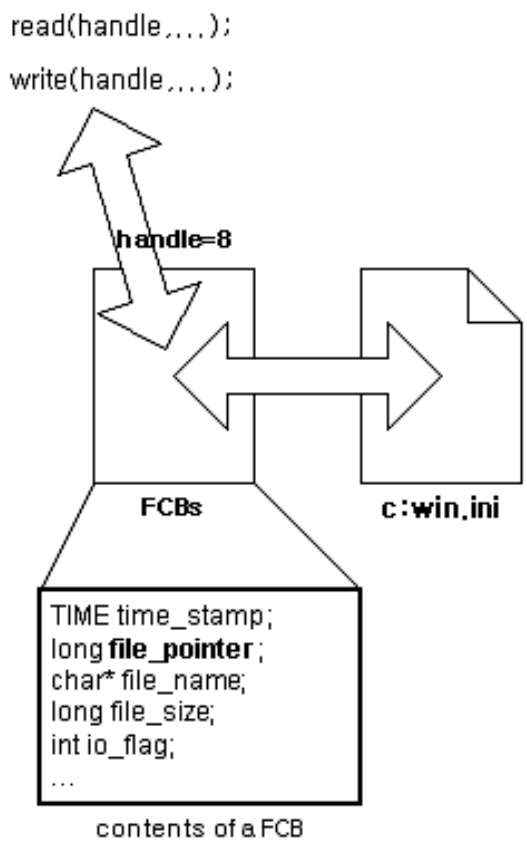
open()의 동작: 디스크에 존재하는 파일의 정보를 새로 할당한 FCB에 채웁니다. 그리고 핸들(이 경우 8)을 리턴합니다. 디스크에 파일이 없거나, FCB를 할당할 공간이 없거나 등의 에러인 경우, open()은 -1(0xFFFF)을 리턴합니다.

• 이제 파일에 관한 정보가 FCB에 채워졌고, 파일 핸들(예를 들면 8)을 알고 있으므로, 파일에 입/출력을 할 수 있습니다.

```
unsigned char c;
read(handle,&c,1);
```

• 예를 들면 위의 예는 파일에서 1바이트를 읽어 c에 저장합니다.
• read()의 파라미터는 처음부터 각각 다음 정보를 인자로 명시해 주어야 합니다.

파일 핸들, 파일에서 내용을 읽을 메모리 주소, 읽을 바이트 크기



read(), write()의 동작: read()와 write()등의 입출력 함수는 첫 번째 파라미터에서 명시된 파일 핸들(8)을 이용하여 실제 디스크의 파일에 입/출력을 합니다.

- 이 프로그램을 구동했을 때, win.ini 파일의 앞 일부분은 다음과 같습니다.

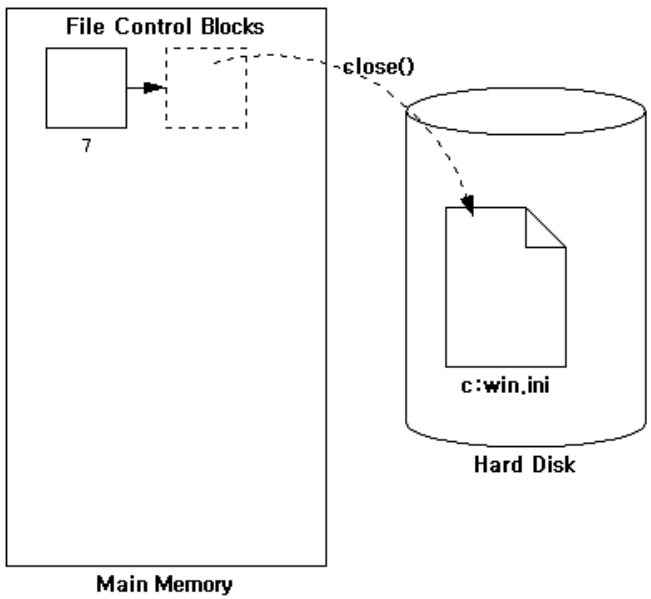
```
[windows]
NullPort=None
ScreenSaveActive=1
ScreenSaveTimeOut=300
SkipMouseRedetect=0
device=HP DeskJet 890C Series,HPRDJC06,LPT1:

[Desktop]
Wallpaper=(없음)
...
```

win.ini파일의 일부분

• 파일에 관한 입출력 작업이 모두 끝났다면, 이제 FCB를 다시 운영체제에게 돌려주어야 합니다.

```
close(handle);
```



close()의 동작: close()는 운영체제에 할당된 FCB를 다른 파일이 사용할 수 있도록 메모리에서 해제합니다. win.ini 파일에 관한 정보는 이제 참조할 수 없습니다.

- 아래의 예를 참고하세요.

```
#include <stdio.h>
#include <string.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>

void main(){
    int handle;
    unsigned char c;

    handle=open("c:\\windows\\win.ini",O_RDWR|O_BINARY);
    if (handle==-1) return;
    read(handle,&c,1);
    printf("%c",c);
    close(handle);
}//main()
```

- read() 호출을 2개 더 추가하여, main()의 소스를 다음과 같이 수정해 보겠습니다.

```
    ...
    if (handle==-1) return;
    read(handle,&c,1);
    printf("%c",c);
    read(handle,&c,1);
    printf("%c",c);
    read(handle,&c,1);
    printf("%c",c);
    close(handle);
}//main()
```

- 출력 결과는 다음과 같습니다.

[wi

파일 포인터(file pointer)

- FCB에 유지되는 중요한 멤버 중 하나가 바로 **파일 포인터(FP: File Pointer)**입니다.
- 파일 포인터는 일반적으로 4바이트 정수로 유지되는데, 파일에서 입/출력 작업이 일어나야 할, 파일의 처음에서 상대주소(relative address)를 가리킵니다.
- 파일을 여는 순간 FP는 0으로 초기화됩니다.

read(handle,&c,1)

"**handle이 가리키는 FCB에서 FP를 읽어서, FP가 가리키는 곳에서 1바이트를 &c로 복사하고, FP를 1증가시킵니다.**"

- read()와 write()등의 함수는 자동으로 FP를 증가시킵니다.

이진 파일 vs. 텍스트 파일

- 예전의 볼런드 컴파일러에서는 표준함수 printf()와 유사한 cprintf() 함수가 있었습니다.
- 볼런드는 자사의 컴파일러에 색을 지원하면서, printf()와 똑 같이 동작하는 아래의 함수들을 추가했습니다.

cprintf(), cscanf()

• 볼런드 컴파일러로 생성한, 아래의 프로그램은 hello를 흰색(WHITE)으로, 다음 줄에 world를 밝은 초록색(LIGHTGREEN)으로 출력하는 소스입니다. 출력 결과에 주목하세요.

```
#include <conio.h>

void main() {
    textcolor(WHITE);
    cprintf("hello\n");
    textcolor(LIGHTGREEN);
    cprintf("world\n");
}
```

• 출력 결과는 다음과 같습니다.

```
hello
_____world
```

고전(oldest), 그러나 ASCII

• 세계 표준 7비트 코드인 **ASCII(American Standard Code for Information Interchange)**
• ASCII는 7비트 코드이며, 표준입니다. 이것은 2^7=128개의 서로 다른 문자를 표현할 수 있으므로, 키보드의 100여 개의 서로 다른 문자를 유일하게 표현할 수 있습니다.
• 아스키코드는 크게 **출력 가능한 문자(printable character)**와 **제어 문자(control character)**로 나눌 수 있습니다.

0~31: 제어 문자
32~126: 출력 가능 문자
127: 제어 문자

• Enter는 13 입니다. 이러한 제어 문자는 화면에 뭔가를 출력하는 것이 아니라, 커서를 다음 줄로 보내거나, 명령을 입력하는 등의 제어를 담당합니다.
• Escape는 27 입니다.

• 32에서 126까지의 출력 가능한 문자의 중요한 아스키코드는 다음과 같습니다.

32: 공백
48: '0'(문자 0)
65: 'A'■
97: 'a'

• 0에서 31사이의 제어문자에는 '데이터 전송의 끝', '데이터 전송의 시작', '파일의 끝', '문자열의 끝' 등을 나타내는 각각의 제어 기능이 예약되어 있습니다.
• 10은 '커서를 다음 줄로'를 의미하는 'New Line' 문자입니다.
• 13번은 '커서를 줄의 처음으로'를 의미하는 'Carriage Return' 문자입니다.
• C++에서 10번과 13번은 문자로 나타낼 수 없으므로, 각각 Escape 절차를 사용하여, '₩n', '₩r'로 나타냅니다.

- Windows 플랫폼에서 키보드의 Enter를 치면, 두 개의 문자가 발생합니다. 그것은 차례대로 13번과 10번 문자입니다.
- 그러므로, 스트링 표현에서 줄을 바꾸는 표현은 다음과 같이 사용하면 잘못된 것입니다.

```
printf("hello₩n");
```

- ₩n만 명시해서는 커서가 줄의 처음으로 이동하지 않습니다.
- 다음과 같이 ₩r과 ₩n을 동시에 명시해 주어야 합니다.

```
printf("hello₩r₩n");
```

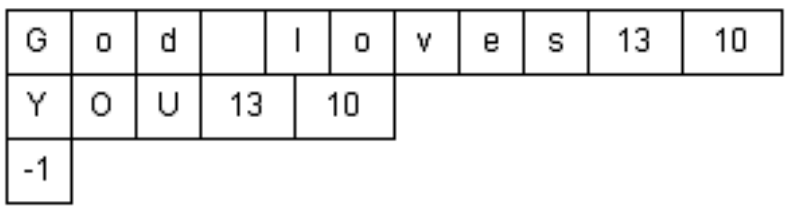
• 그런데 왜 printf()등의 함수에서는 ₩n이 줄을 바꾸는 역할을 하는 것일까요? 그것은 바로 다른 운영 체제(예를 들면 UNIX)와의 차이점 때문입니다.
• UNIX에서는 줄을 끝을 나타내는 문자는 '₩n'(10)입니다. 하지만, 도스(DOS)에서는 '₩r₩n'(13 10)입니다. 하지만, printf()는 표준함수이므로 일관된 표현을 가져야 합니다.
• 그러므로, DOS 운영체제의 컴파일러는 printf()등의 표준 함수에 사용된 "₩n"을 "₩r₩n"이 되도록 자동으로 코드를 생성합니다.
• 하지만, cprintf()등은 표준 함수가 아니므로, 프로그래머가 명시적으로 "₩r₩n"을 붙여주어야 하는 것입니다.

• 이제 아스키의 특수문자가 파일에 포함되었다고 가정해 봅시다. 그렇다면, 파일에 있는 0에서 31사이의 제어 문자를 어떻게 해석해야 하는가요?
• 제어 문자로 해석하면, 이것을 **텍스트 파일(text file)**이라고 합니다. 데이터로 해석하면, 이것은 **이진 파일(binary file)**이라고 합니다.

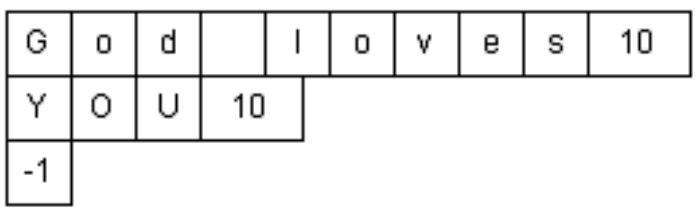
- 파일을 이진 파일로 여는 것과, 텍스트 파일로 여는 것에는 어떤 차이가 있을까요?
- 그것은 표준 함수(fscanf(), fprintf()등)가 동작하는 방식을 결정합니다.
- Windows 플랫폼에서 파일을 텍스트 파일로 열었을 경우 문자열의 끝에 있는 13은 읽히지 않습니다. 이것은 UNIX와의 호환을 위해서 문자열의 끝에는 10이 있다고 가정하기 때문입니다.
- 하지만, 이진 파일로 열었을 경우 문자열의 끝에 있는 13,10이 모두 읽힙니다.

God loves
YOU

- 위의 텍스트 파일은은 DOS와 UNIX에서 각각 아래 그림처럼 저장됩니다.



DOS file system



UNIX file system

- 위의 텍스트 파일을 data.txt라고 합시다. 이제 이 파일을 텍스트 모드로 엽니다.

```
char s[80]
FILE* fp=fopen("data.txt","r");
fscanf(fp,"%s",s);
```

- 그러면 DOS/Windows 플랫폼에서 위 문장은 s에 다음의 값을 저장합니다.

God loves₩n

- 하지만, data.txt를 이진 파일로 열었을 경우는 다르게 동작합니다.

```
char s[80];
int handle=open("data.txt",O_BINARY);
read(handle,s,10);
```

- 위 소스는 DOS와 UNIX 모두의 경우 10바이트만 읽으므로, DOS/Windows에서 생성한 텍스트 파일인 경우는 아래의 값이 읽힙니다.

```
God loves\r
```

- 하지만, UNIX에서 만든 텍스트 파일인 경우는 다음의 값이 읽힙니다.

```
God loves\n
```

텍스트 파일의 처리

• 텍스트 파일을 2진 파일로 열어서 줄(line)을 읽고, 왼쪽과 오른쪽의 공백을 제거하는 프로그램을 구현해 봅시다.

• ReadLine()은 한 줄을 읽습니다.

```
int ReadLine(int handle, char s[]){
    char ch=0;
    int i=0;

    if (eof(handle)) return -99;
    while (ch!=10 && !eof(handle)){
        read(handle,&ch,1);
        s[i++] = ch;
    }//while
    s[i]=0;//stuff EOS mark
    return i-2;//return length of the string
}//ReadLine()
```

• 아래의 소스는 win.ini 파일의 내용을 모두 출력합니다.

```
#include <stdio.h>
#include <string.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>

int ReadLine(int handle, char s[]){
    char ch=0;
    int i=0;

    if (eof(handle)) return -99;
    while (ch!=10 && !eof(handle)){
        read(handle,&ch,1);
        s[i++] = ch;
    }//while
    s[i]=0;//stuff EOS mark
    return i-2;//return length of the string
```

```
}//ReadLine()

int LTrim(char s[]){
    int i=0;
    char ch=s[i];
    while (ch==' '){
        ch=s[++i];
    }//while
    memmove(&s[0],&s[i],strlen(s)-i+1);
    return i;//return the number of removed blanks
}//LTrim()

int RTrim(char s[]){
    int i=strlen(s)-1;
    int count=0;
    char ch;
    ch=s[i];
    while (ch==' '){
        ch=s[--i];
```

```
        count++;
    }//while
    s[i+1]=0;
    return count;//return the number of removed blanks
}//RTrim()

char* Trim(char s[]){
    LTrim(s);
    RTrim(s);
    return s;
}//Trim()

void main(){
    int handle;
    handle=open("c:\\windows\\win.ini",O_RDWR);
    if (handle==-1)
        return;
    char s[300];
    while (ReadLine(handle,s)!= -99){
```

```
        printf(Trim(s));
    }//while
    close(handle);
}//main()
```

이진 파일의 처리

- 이진 파일을 열어서 내용을 모두 콘솔화면에 출력하려고 합니다.
- 이진 파일에 포함된 특수문자는 화면에 출력할 수 없습니다. 그래서 읽은 값이 특수문자이면 점(.) 을 출력합니다.
- 화면에 출력할 수 있는 문자이면 그 문자를 그대로 출력합니다.

```
#include <stdio.h>
#include <ctype.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>

void main(){
    int handle;
    unsigned char c = 0;
    char s[80];
```

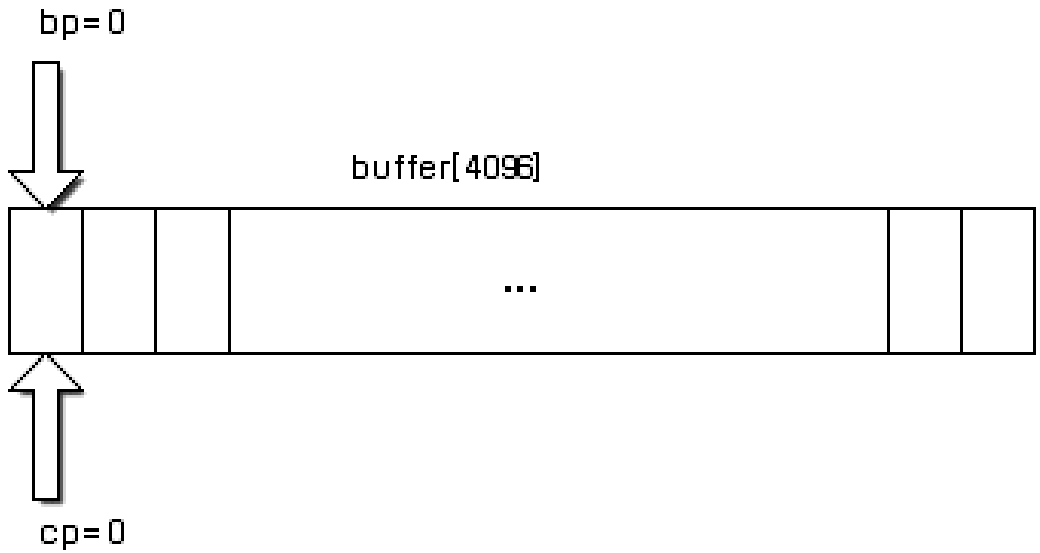
```
    int state = 0;

    handle = open("C:\\windows\\system32\\cmd.exe", O_BINARY);
    //          ^command.com의 적절한 경로를 명시해야 한다.
    if (handle == -1) return;//에러가 발생한 경우
    while (!eof(handle)){
        read(handle, &c, 1);
        if (iscntrl(c))//0~31사이의 문자인가?
            printf(".");
        else
            printf("%c", c);
    }//while
    close(handle);
}//main
```

버퍼링(buffering)

- 버퍼링이란 디스크 접근 시간을 줄이기 위해서, 프로그램에서 임시로 만든 버퍼(buffer)에, 파일로부터 한꺼번에 많은 양의 데이터를 읽어들이는 방식을 말합니다.
- 일반적으로 버퍼의 크기는 2의 n승(2 to the power of n)으로 설정합니다.

• 버퍼의 크기를 4096(2^{12}), 버퍼로 읽어 들인 크기를 bp, 버퍼에서 현재 위치를 cp가 가리킨다고 합시다. 프로그램의 초기 상태는 bp=cp=0에서 시작합니다.

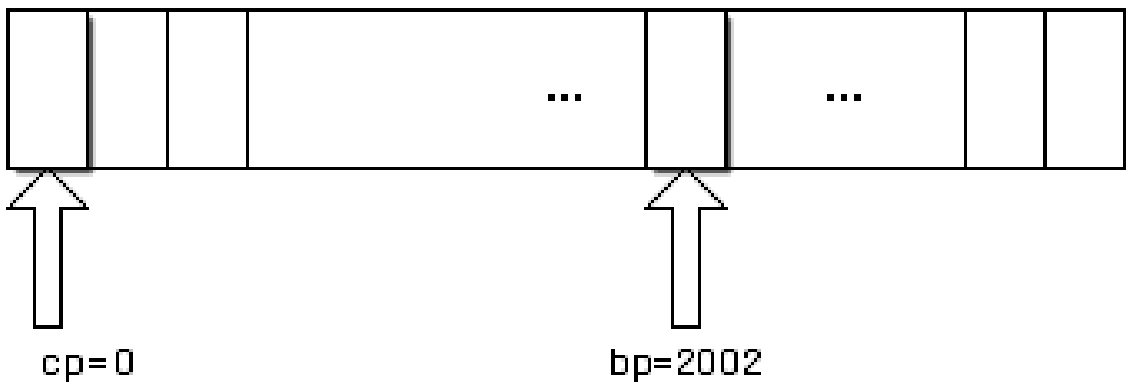


버퍼의 초기 상태: bp=cp=0이거나, cp>=bp가 될 때 버퍼는 새로운 내용으로 갱신되어야 합니다.

• 프로그램에서 파일의 4096바이트를 읽으려고 시도할 때마다, 대부분의 경우에, 마지막 경우를 제외하고 4096바이트가 모두 읽힐 것입니다.

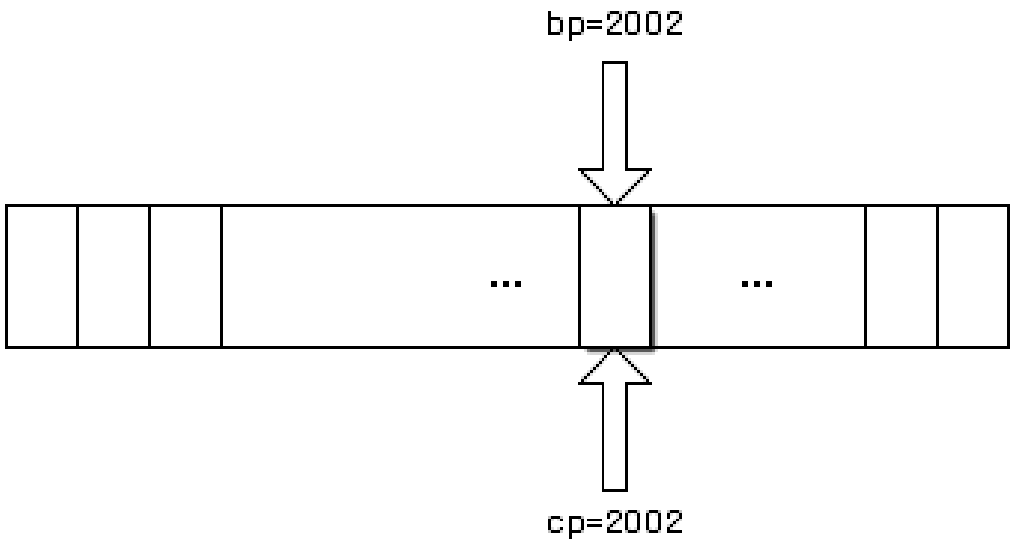
버퍼가 꽉 찬 상태: 대부분의 경우에 버퍼에는 4096바이트가 모두 찹니다(full).

• 하지만, 파일의 마지막 부분에서는 4096바이트가 아닌 다른 값이 될 확률(4095/4096)이 높습니다. 예를 들면, 마지막에 파일에서 2002바이트를 읽었다고 합시다.



파일의 마지막을 읽었을 때 버퍼의 상태: 파일의 마지막에서는 bp가 4096이 되지 않을 확률이 높습니다. 이 값을 2002라고 합시다.

• 몇 바이트가 버퍼에 들어있던지, cp가 bp를 넘어갈 수는 없습니다. cp가 bp를 넘어가는 조건은 파일에서 버퍼로 데이터를 읽어야 함을 의미합니다.



버퍼의 갱신 상태: 현재 버퍼가 꽉 찼던(bp=4096), 꽉 차지 않았던(bp=2002), cp가 bp와 같아진다면, 버퍼의 내용을 갱신해야 합니다.

• 같은 소스를 버퍼링을 사용한 예를 아래에 리스트하였습니다.

```
#include <stdio.h>
#include <ctype.h>//for 'iscntrl()'
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>

unsigned char buffer[4096];
int bEof=0,//EOF flag
    bp=0,//buffer size pointer
    cp=0;//current pointer

void ReadToBuffer(int handle) {
    cp=0;
    if (bp>0 && bp<4096) {//bp가 4096보다 작다면, 더 이상 읽어서는 안된
                          //다. bp>0 조건은 시작 상태를 구분하기 위해 사
                          //용하였다.
        bEof=1;
```

```
        return;
    }//if
    bp=read(handle,buffer,4096);//read는 읽은 바이트 수를 리턴한다.
}//ReadToBuffer

unsigned char ReadChar(int handle) {
    if (cp>=bp)//버퍼를 갱신해야 한다.
        ReadToBuffer(handle);
    return buffer[cp++];
}//ReadChar

void main(){
    int handle;
    unsigned char c;
    char s[80];
    int state=0;

    handle=open("c:\\windows\\system32\\cmd.exe",O_BINARY);
    if (handle==-1) return;
```

```
    while (1){
        c=ReadChar(handle);
        if (bEof) break;
        if (iscntrl(c))
            printf(".");
        else
            printf("%c",c);
    }//while
    close(handle);
}//main
```

실습문제

1. MFC(Microsoft Foundation Class)에서 지원하는 CFile과 같은 기능을 하는 클래스를 제작하세요.

2. 파일 입출력을 지원하는 풀 스크린 편집기(full screen editor)를 작성하세요.

3. win.ini파일을 출력하는 소스 코드를 버퍼링을 지원하도록 수정하세요.

4. 버퍼링을 구현한 소스에서 ReadChar()함수는 오직 1바이트를 읽습니다. 여러 바이트를 읽는 ReadData()함수를 추가하세요. 함수의 원형은 다음과 같습니다.

```
int ReadData(int handle,void* data, int size)
```

@