

삼성 청년 SW 아카데미

임베디드 C언어

<알림>

본 강의는 삼성 청년 SW아카데미의 콘텐츠로
보안서약서에 의거하여
강의 내용을 어떠한 사유로도 임의로 복사,
촬영, 녹음, 복제, 보관, 전송하거나
허가 받지 않은 저장매체를
이용한 보관, 제3자에게 누설, 공개,
또는 사용하는 등의 행위를 금합니다.

Day6-1. 포인터 응용

챕터의 포인트

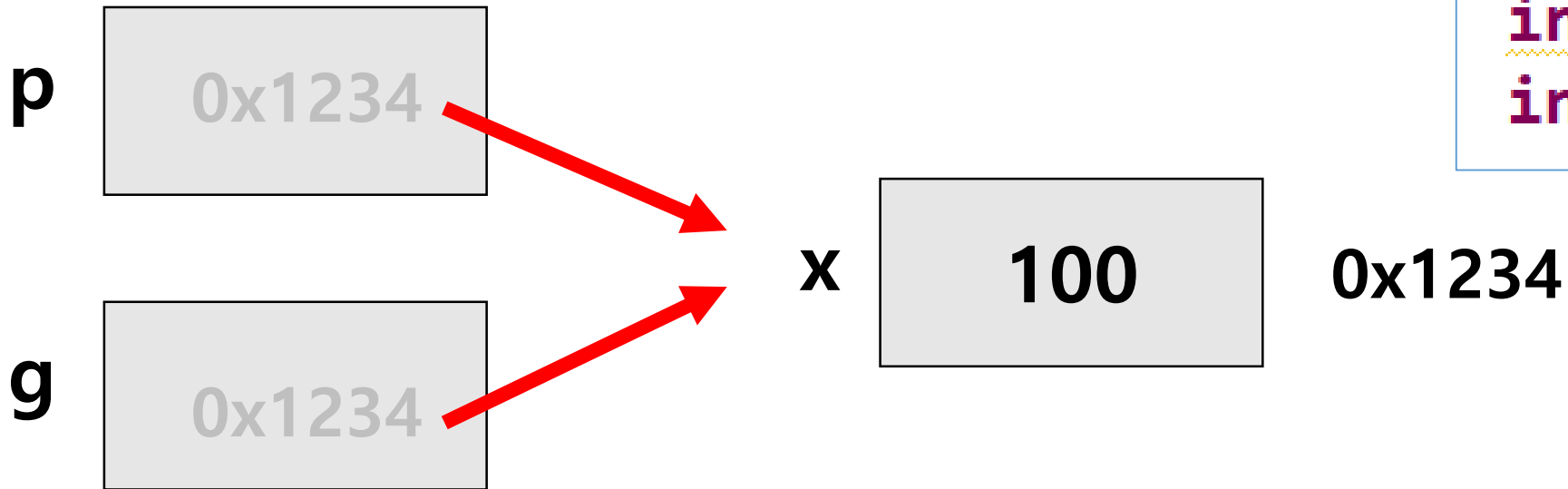
- void 포인터
- 포인터 배열
- 배열 포인터
- 함수 포인터
- 함수 포인터 배열
- 구조체와 함수포인터

포인터 복습

목표

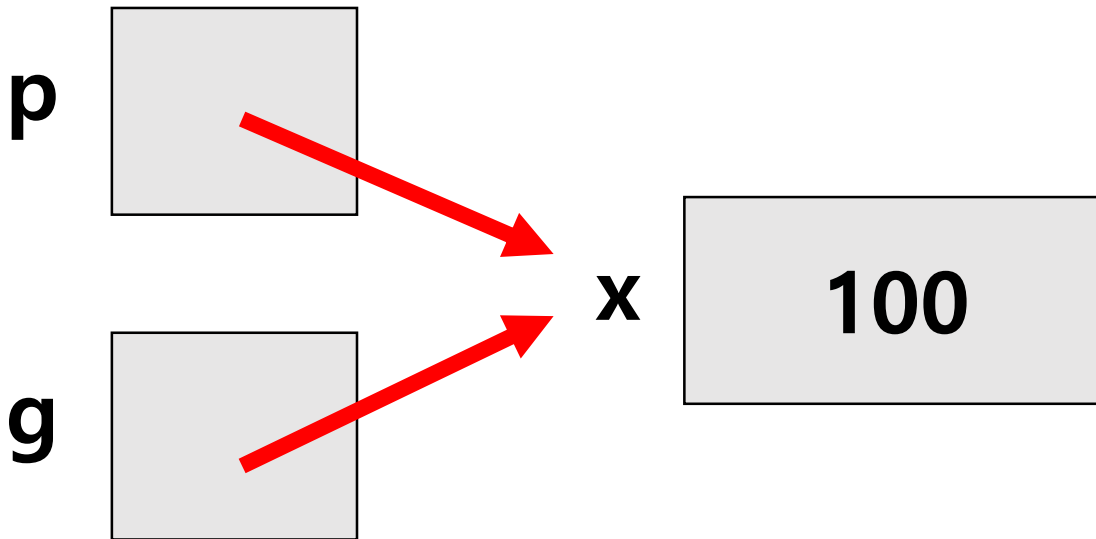
- 포인터를 다시 상기한다.
- 포인터를 이용해 변수의 값을 바꿀 수 있다.

포인터가 주소를 저장할 때,
“가리킨다” 라고 표현한다. (화살표로 표현 가능)



```
int x = 100;  
int *p = &x;  
int *g = &x;
```

포인터로 변수를 가리키고
값을 제어할 수 있다.

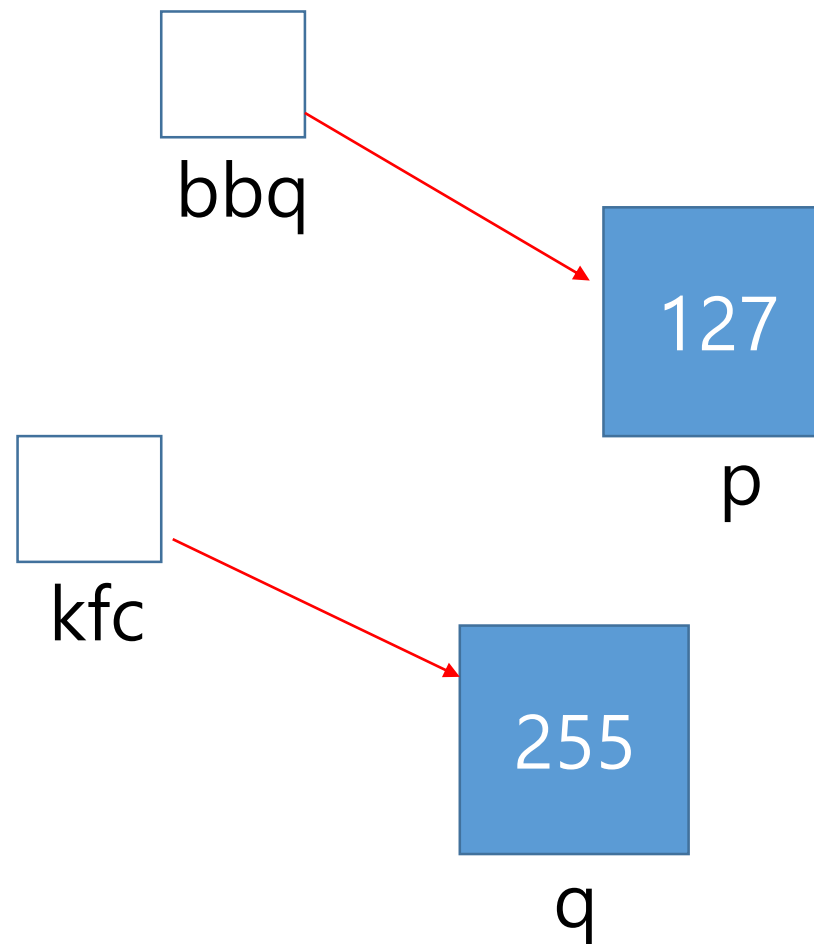


```
int x = 100;  
int* p = &x;  
int* q = &x;  
  
*p = 5000;  
  
printf("%d", x);
```


2개의 변수를 생성하고

각각을 포인터 변수로 가리킨다.

- 포인터 변수를 이용해서 각 변수의 값을 바꿔서 출력한다.



void 포인터

목표

- void 포인터에 대해 학습한다.
- 모든 형태의 포인터를 다 받아준다.

일반적인 포인터 사용 방법

- int 변수를 가리키는 포인터의 타입 = int *
- char 변수를 가리키는 포인터의 타입 = char *

같은 타입의 포인터 변수를 만들어서 가리킨다.

```
#include<stdio.h>

int main() {

    int a = 10;
    char b = 'A';

    int* pa = &a;
    char* pb = &b;

    return 0;
}
```

void 포인터는 만능 포인터이다.

- 어떤 타입의 주소도 모두 다 저장할 수 있는 만능 포인터
- 모든 것의 신이다

```
#include<stdio.h>

✓int main() {

    int a = 10;
    char b = 'A';

    void* pa = &a;
    void* pb = &b;

    return 0;
}
```

void 포인터의 기능 : 주소 저장만 가능

- 주소를 저장할 수 있다.
- 하지만, 사용할 순 없다. 저장만 할 수 있다.

```
void * p1 = &x;  
void * p2 = &t;  
  
printf("%d", *p1);
```

void 포인터로 가리킨 값은
형변환을 해줘야 한다.

<https://gist.github.com/hoconoco/98cd94e675fe93993c55eb0d090c4c91>

```
#include<stdio.h>

int main() {

    int a = 10;
    char b = 'A';

    void* pa = &a;
    void* pb = &b;

    printf("%d\n", *(int*)pa);
    printf("%c\n", *(char*)pb);

    return 0;
}
```

함수의 매개변수로도 사용 가능하다

<https://gist.github.com/hoconoco/2ee7fff40015d990a0161a7d0dbe74a9>

```
#include <stdio.h>

void abc(void* v){
    printf("%d\n", *(int*)v);
}

int main(){
    int x = 10;
    abc(&x);
    return 0;
}
```


구조체도 받을 수 있다!

```
void abc(void* v){
    struct Node val = *(struct Node*)v;
    printf("%d %d\n", val.y, val.x);
}

int main(){
    struct Node bbq = { .y = 10, .x = 20 };
    abc(&bbq);
    return 0;
}
```

<https://gist.github.com/hoconoco/45bed94fe62f3025198397ffeb25be46>

포인터 배열

목표

- 포인터 배열에 대해 학습하고 이해한다.

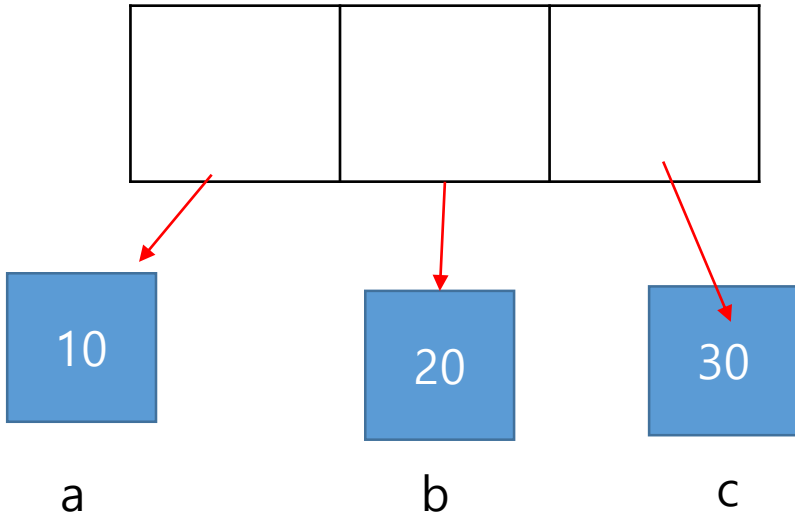
포인터가 여러 개이다.

- 포인터 꾸러미
- 각각의 요소들을 가리킨다.

```
int* p[3] = {NULL, NULL, NULL};
```



그림과 같이 가리키기



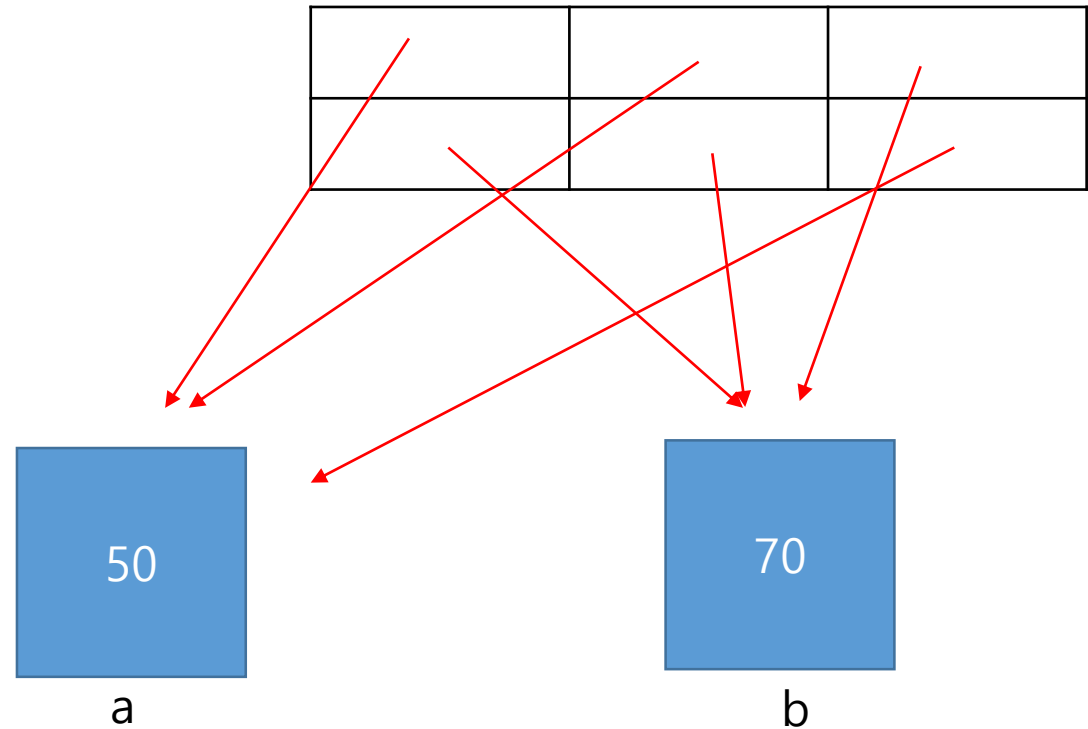
```
int a = 10;  
int b = 20;  
int c = 30;  
  
int* p[3] = { &a, &b, &c };
```

<https://gist.github.com/hoconoco/89d7bd2743541173f5946b195b11e025>

다음 그림과 같이 가리킨다.

2중 for를 돌려 출력한다.

```
Microsoft Visual C++  
50 50 70  
70 70 50
```



배열 포인터

목표

- 배열을 가리키는 포인터인 배열 포인터에 대해 학습하고 이해한다.

사과맛 딸기는

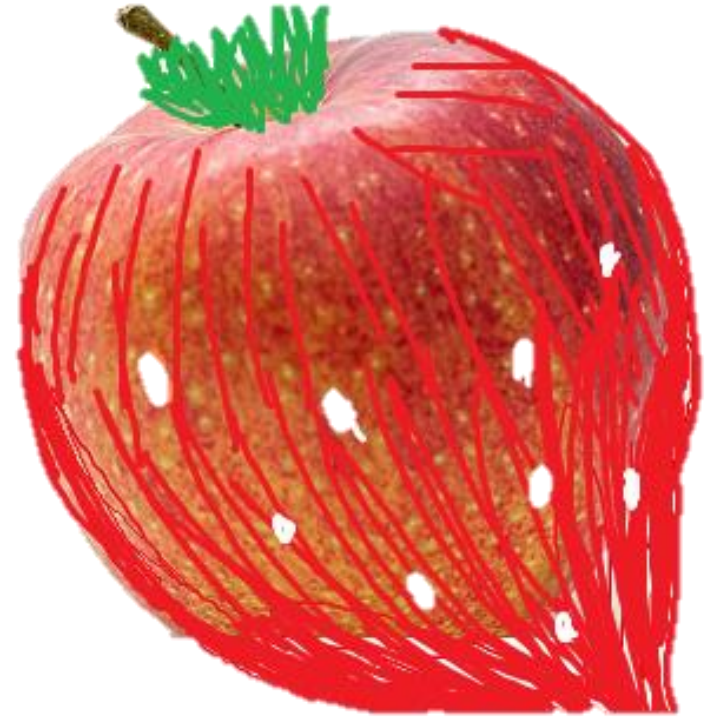
1. 사과일까?
2. 딸기일까?

사과맛 딸기는

1. 사과일까?
2. 딸기일까?

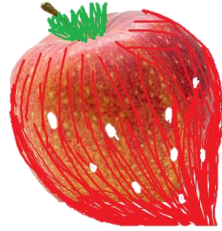
그럼 배열 포인터는

1. 포인터이다?
2. 배열이다?



맛있는 사과맛 딸기

배열을
가리키는 포인터이다.



```
int arr[3] = { 10, 20, 30 };
```

```
int(*p)[3];  
p = arr;
```

배열의 이름은 배열의 시작 메모리 주소 값이다.

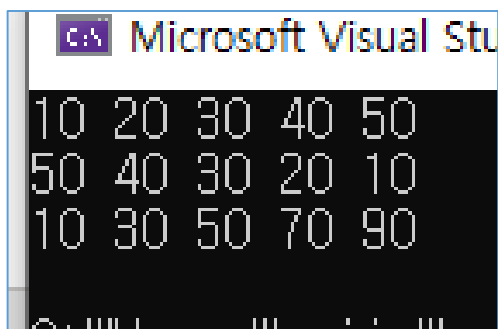
- `p = arr`
- `p = &arr`
- 모두 같은 표현이다.

```
int arr[3] = { 10, 20, 30 };  
  
int(*p)[3];  
p = arr;
```

<https://gist.github.com/hoconoco/0619b5297ae62f2dc682f3188df4b5dd>

배열 포인터를 이용해
2차원 배열을 가리켜보자.

- `int (*p)[3][5];`



```
C++ Microsoft Visual Studio
10 20 30 40 50
50 40 30 20 10
10 30 50 70 90
```

```
int arr[3][5] = {
    {10, 20, 30, 40, 50},
    {50, 40, 30, 20, 10},
    {10, 30, 50, 70, 90}
};
```

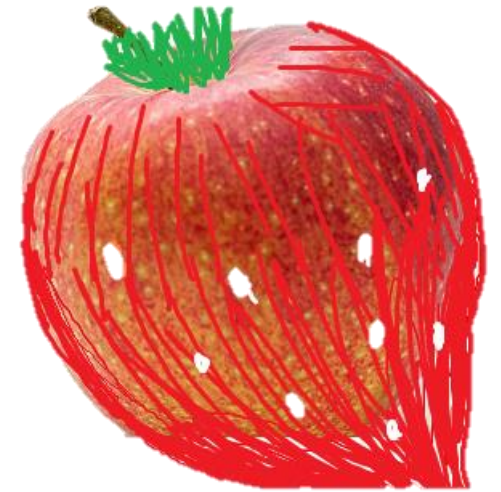
함수 포인터

목표

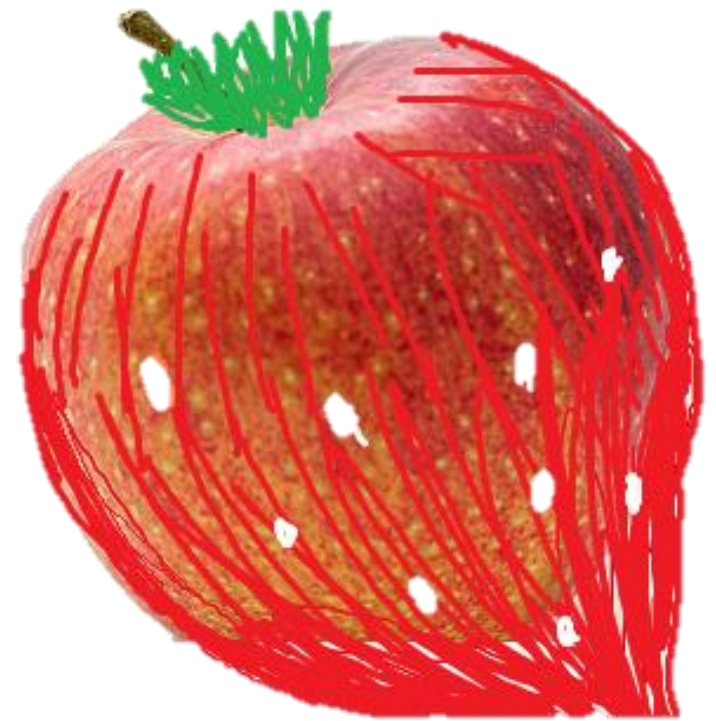
- 함수를 가리키는 포인터인 함수 포인터에 대해 학습하고 이해한다.

목표

- 함수를 가리키는 함수 포인터에 대해 학습한다.



함수를 가리키는 포인터!



함수의 이름도 주소이다.

- 함수에 매개변수가 있다면, 타입을 맞추는 게 좋다.

<https://gist.github.com/hoconoco/1d28afea0a3beda359e2fbcf2c5a3e4d>

```
void abc(int a) {  
    printf("ABC %d\n", a);  
}  
  
int bbq() {  
    printf("BBQ\n");  
    return 10;  
}  
  
int main() {  
  
    void (*p)(int);  
    p = abc;  
    (*p)(10);  
  
    int (*q)();  
    q = bbq;  
    printf("%d", (*q)()+100);  
}
```

함수 포인터 배열

목표

- 함수 여러 개를 가리키는 함수 포인터 배열에 대해 학습하고 이해한다.

바로 확인해보자.



```
void (*p[5])(); //함수포인터 배열
```

함수 포인터 배열은 모두 같은 형식이어야 한다.

- 매개변수가 모두 있거나, return 타입 일치

```
int abc(int n) {  
    printf("ABC %d\n", n);  
    return 321;  
}  
  
int bbq(int n) {  
    printf("BBQ %d\n", n);  
    return 123;  
}  
  
int main() {  
  
    int (*p[2])(int) = { abc, bbq };
```

<https://gist.github.com/hoconoco/3ea757959a6768e30119263205984aa9>

구조체와 함수 포인터

목표

- 함수 포인터를 멤버로 갖는 구조체의 사용법을 이해하고 실습한다.

여러 타입을 묶어주는 타입

함수 포인터를 멤버로 갖는 구조체도 있다!

- 디바이스 드라이버 개발에 사용할 file_operations 구조체

```
struct file_operations {  
    struct module *owner;  
    loff_t (*llseek) (struct file *, loff_t, int);  
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);  
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

<https://github.com/raspberrypi/linux/blob/rpi-6.1.y/include/linux/fs.h#L2152>

디바이스 드라이버 개발할 때
사용하는 방식을 미리 체험한다.

```
int main() {  
    // 구조체 변수 선언 및 선택적 초기화  
    struct ssafy_opeartions fops = {  
        .levelup1 = BBQ,  
        .levelup2 = ABC,  
    };  
  
    // 함수 포인터를 멤버로 갖는 구조체의 함수 호출  
    fops.levelup1(10); // BBQ 호출  
    fops.levelup2(20); // ABC 호출  
}
```

<https://gist.github.com/hoconoco/104b984ce39e92f50b46b783170fdbe3>

Day6-2. 변수 keyword

챕터의 포인트

- const
- volatile
- main argument

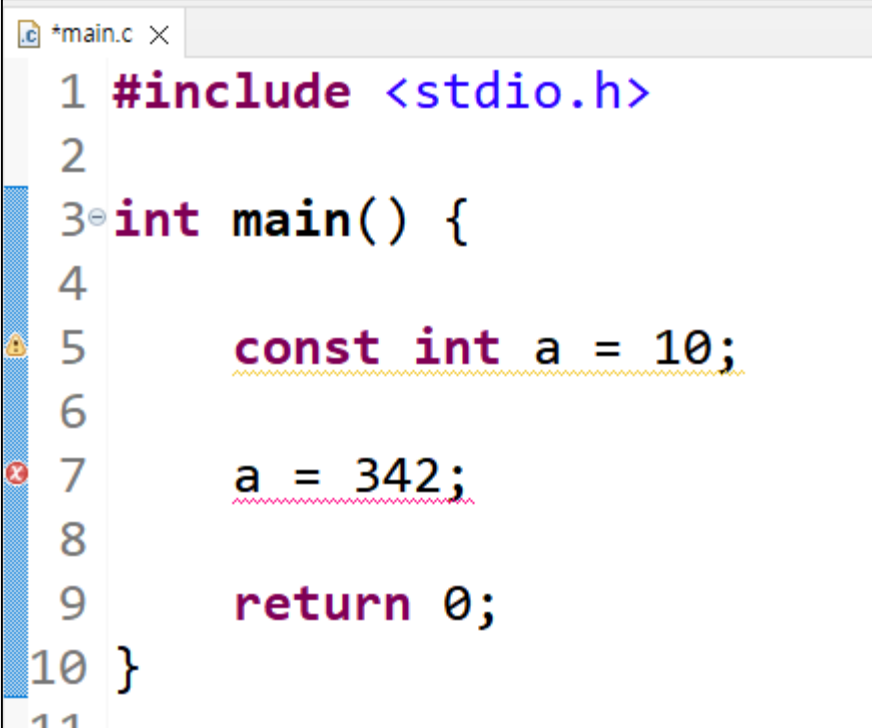
const

목표

- `const` 한정사의 쓰임에 대해 학습한다.

const = 상수

- 값을 바꿀 수 없음
- 이 무지막지한 키워드는 어느 곳이나 붙어 있다.
- C++ 에서는 더 난리가 난다.
- 수정할 수 없어서 에러가 난다.



```
*main.c x
1 #include <stdio.h>
2
3 int main() {
4
5     const int a = 10;
6
7     a = 342;
8
9     return 0;
10 }
11
```

에러 발생

const 의 문제는 순서도 바뀌서 붙는 다는 것이다.

- 매 번 힘들게 한다.

```
4  int x = 10;
5  const int *p = &x;
6  printf("%d\n", *p);
7
8  //값 변경 불가
9  *p = 100;
10
11 printf("%d\n", *p);
```

자주 쓰이는 형태

가리킬 대상은 바꿀 수 있지만,
가리킨 곳의 값을 바꿀 수 없음

```
4  int x = 10;
5  int* const p = &x;
6
7  //값은 변경 가능
8  *p = 1000;
9  printf("%d ", x);
10
11 //다른 변수 가리키기 불가능
12 int y = 20;
13 p = &y;
```

자주 안쓰임

가리킨 곳의 값은 변경할 수 있지만,
가리킬 대상을 바꿀 수는 없다.

const는 개발을 할 때 우리를 굉장히 성가시게 하지만,
프로그램 측면에서는 굉장히 많은 도움을 준다.

- 프로그램의 안정성 보장
- 가독성 향상
- 컴파일러 최적화
- 디버깅
- 인터페이스 정의 등등

적절하게 const를 활용하는 것은 좋은 프로그래밍 습관이 된다.

volatile

최적화를 방해하는 키워드!

누가 최적화를 하는가?

→ 컴파일러

컴파일러를 방해하는 키워드

- 벌써 신이 난다.

어떻게 동작하는 코드일까?

<https://gist.github.com/hoconoco/e15d422ee6e66aab0b97aa87e551e530>

```
#include<stdio.h>

int main() {
    int a = 10;
    a = 20;
    a = 30;
    a = 40;
    printf("%d",a);
    return 0;
}
```

사람이 짠 .c 코드를 보고 기계어로 변환한다.

```
#include<stdio.h>

int main() {
    int a = 10;
    a = 20;
    a = 30;
    a = 40;
    printf("%d",a);
    return 0;
}
```



0110100011010101010..

컴파일러는 생각보다 똑똑하다.

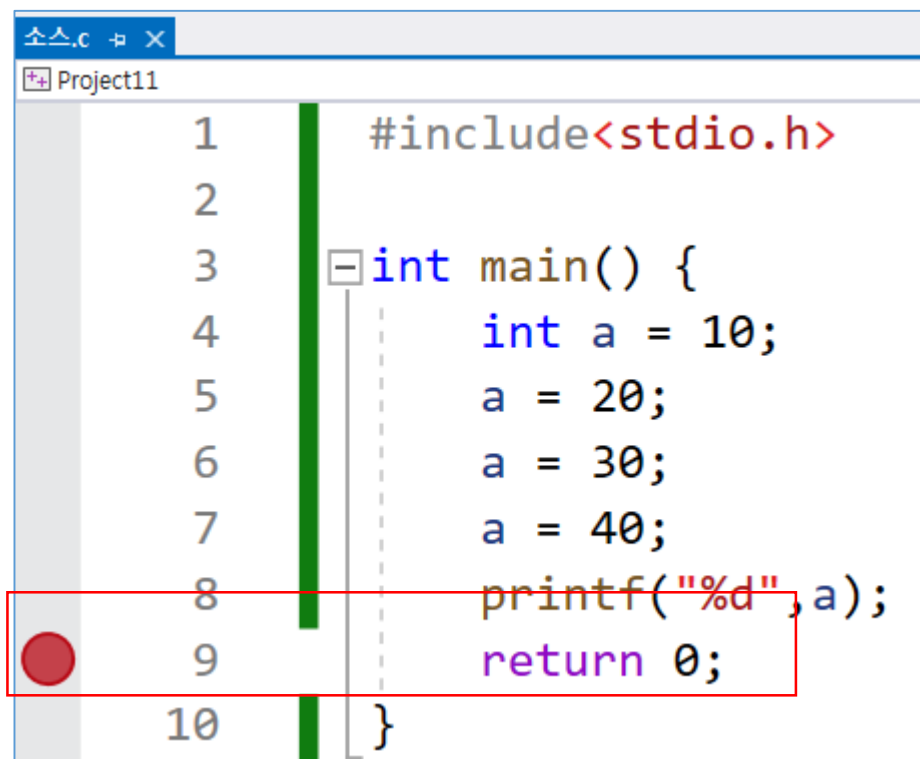
- (코드는 생각보다 길다)
- 컴파일러도 조금만 일하고 싶다.

```
#include<stdio.h>

int main() {
    int a = 10;
    a = 20;
    a = 30;
    a = 40;
    printf("%d",a);
    return 0;
}
```

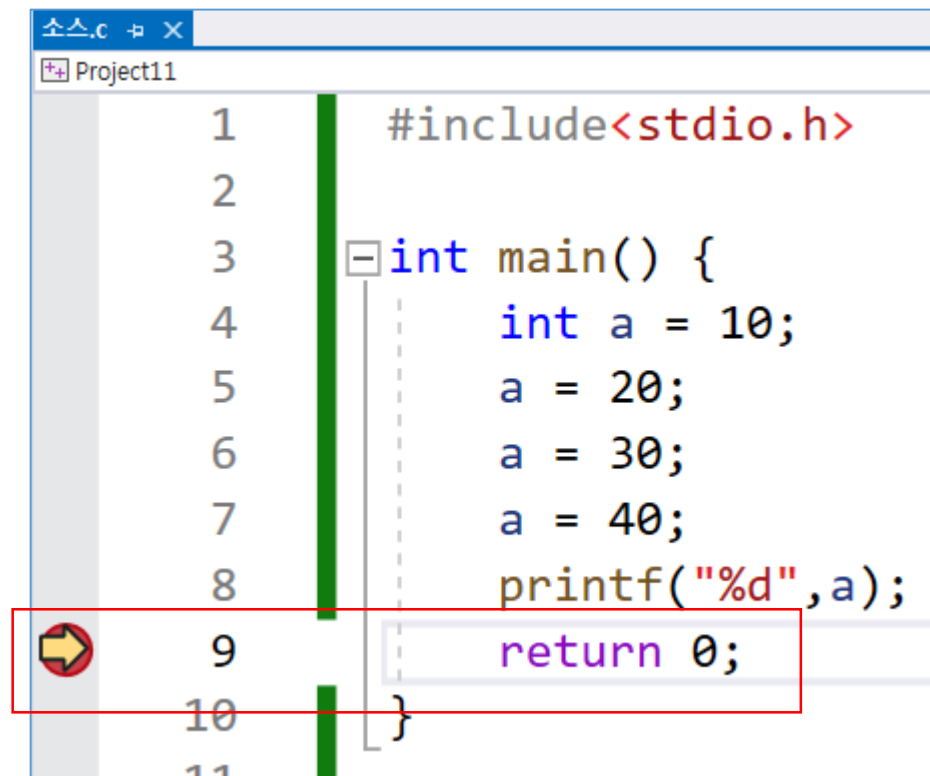
??? : 어차피 40인데 중간에 바뀌는 거 꼭 해야함?

return 에 break point (F9) 걸고,
디버깅(F5) 하자.



```
소스.c ㅁ x
Project11
1  #include<stdio.h>
2
3  int main() {
4      int a = 10;
5      a = 20;
6      a = 30;
7      a = 40;
8      printf("%d", a);
9      return 0;
10 }
```

A red circle breakpoint is placed on the `return 0;` line (line 9) of the `main` function.

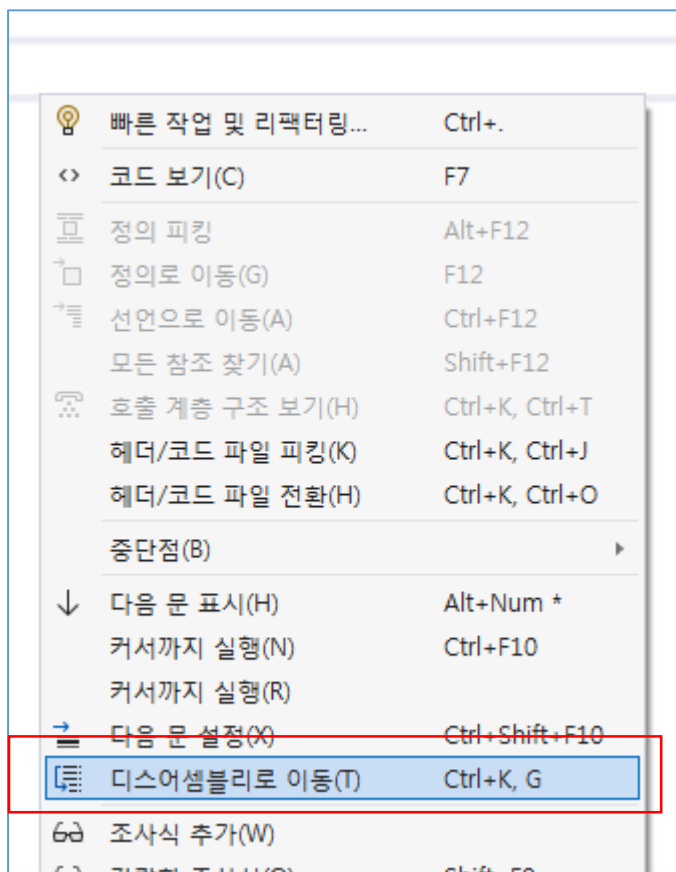


```
소스.c ㅁ x
Project11
1  #include<stdio.h>
2
3  int main() {
4      int a = 10;
5      a = 20;
6      a = 30;
7      a = 40;
8      printf("%d", a);
9      return 0;
10 }
```

A yellow arrow breakpoint is placed on the `return 0;` line (line 9) of the `main` function.

마우스 우클릭해서 디스어셈블리로 이동 클릭

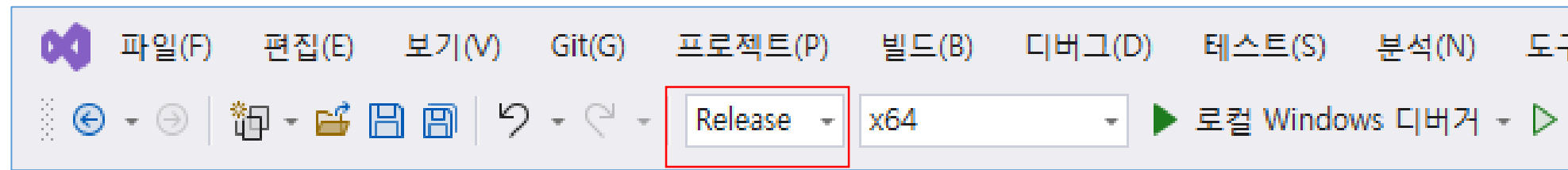
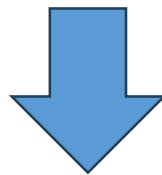
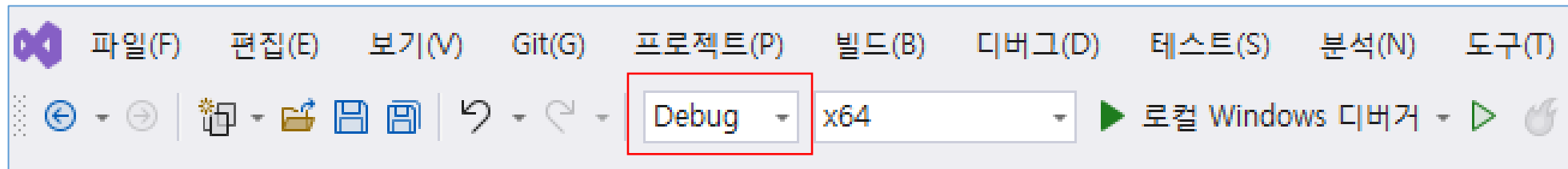
- **mov** : 어셈블리어에서 데이터 복사를 담당하는 명령어



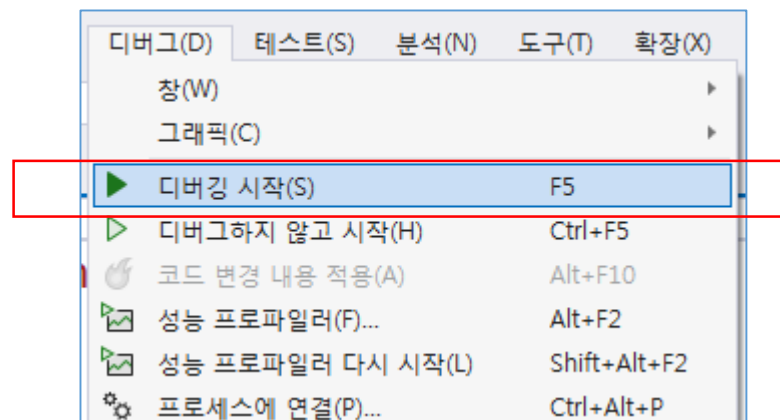
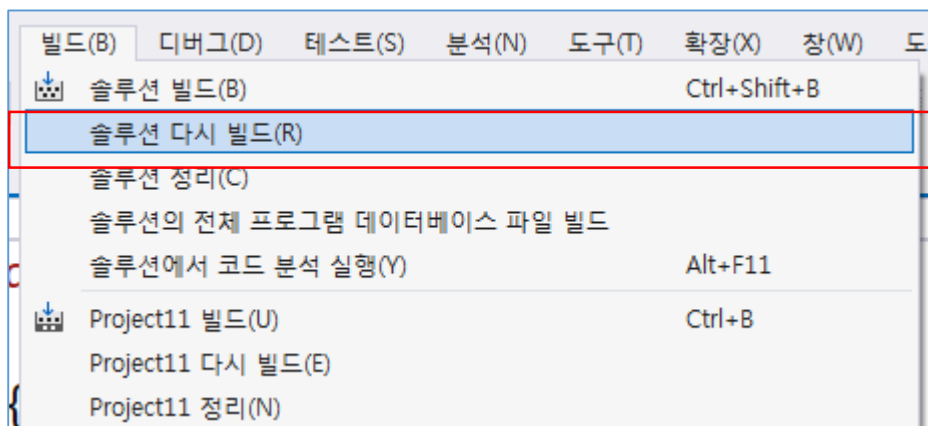
```
00007FF7C2E62AE6  call      __CheckForDebuggerJustMyCode (07FF7C2E61343h)
               int a = 10;
00007FF7C2E62AEB  mov       dword ptr [a],0Ah
               a = 20;
00007FF7C2E62AF2  mov       dword ptr [a],14h
               a = 30;
00007FF7C2E62AF9  mov       dword ptr [a],1Eh
               a = 40;
00007FF7C2E62B00  mov       dword ptr [a],28h
               printf("%d",a);
00007FF7C2E62B07  mov       edx,dword ptr [a]
00007FF7C2E62B0A  lea       rcx,[string "%d" (07FF7C2E69BD8h)]
00007FF7C2E62B11  call      printf (07FF7C2E613A7h)
               return 0;
00007FF7C2E62B16  xor       eax,eax
}
```


디버깅을 멈춘 뒤

Debug → Release로 바꿔보자



솔루션 다시 빌드를 한 뒤
디버깅을 한다.

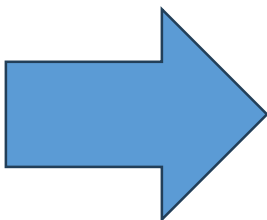


컴파일러는 release 를 할 때,
불필요한 절차를 자멋대로 다 생략하고 최적화 해서 컴파일한다.

- mov 가 한번만 쓰였다.
- 똑똑하다.

```
00007FF7C2E62AE6 call    __CheckForDebuggerJustMyCode (07FF7C2E61343h)
    int a = 10;
00007FF7C2E62AEB mov     dword ptr [a],0Ah
    a = 20;
00007FF7C2E62AF2 mov     dword ptr [a],14h
    a = 30;
00007FF7C2E62AF9 mov     dword ptr [a],1Eh
    a = 40;
00007FF7C2E62B00 mov     dword ptr [a],28h
    printf("%d",a);
00007FF7C2E62B07 mov     edx,dword ptr [a]
00007FF7C2E62B0A lea     rcx,[string "%d" (07FF7C2E69BD8h)]
00007FF7C2E62B11 call    printf (07FF7C2E613A7h)
    return 0;
00007FF7C2E62B16 xor     eax,eax
}
```

debug



```
int main() {
00007FF68F2A1070 sub     rsp,28h
    int a = 10;
    a = 20;
    a = 30;
    a = 40;
    printf("%d",a);
00007FF68F2A1074 mov     edx,28h
00007FF68F2A1079 lea     rcx,[string "%d" (07FF68F2A2250h)]
00007FF68F2A1080 call    printf (07FF68F2A1010h)
    return 0;
}
```

release

변수 타입 앞에 volatile을 붙이고 다시 디버깅하자

```
#include<stdio.h>

int main() {
    volatile int a = 10;
    a = 20;
    a = 30;
    a = 40;
    printf("%d",a);
    return 0;
}
```

volatile 은 컴파일러가 최적화 하는 걸 방해한다.

- mov 명령어가 4번 쓰였다.

• ~~ㅋㅋㅋㅋㅋㅋ~~

```
int main() {
00007FF6136C1070  sub             rsp,28h
    volatile int a = 10;
00007FF6136C1074  mov             dword ptr [rsp+30h],0Ah
    a = 20;
    a = 30;
    a = 40;
    printf("%d",a);
00007FF6136C107C  lea             rcx,[string "%d" (07FF6136C2250h)]
00007FF6136C1083  mov             dword ptr [a],14h
00007FF6136C108B  mov             dword ptr [a],1Eh
00007FF6136C1093  mov             dword ptr [a],28h
00007FF6136C109B  mov             edx,dword ptr [a]
00007FF6136C109F  call            printf (07FF6136C1010h)
    return 0;
00007FF6136C10A4  xor             eax,eax
}
```

임베디드에서 H/W의 특정 주소 값을 이용해 제어하는 경우

- H/W가 연결된 메모리 주소의 값은 사용자가 값을 수정하지 않아도 H/W에 의해 바뀔 수 있다.
- 아래의 코드는 값이 바뀌면 while문을 동작시키는 코드인데, 컴파일러 입장에서는 어차피 바뀔 리가 없다고 생각해 최적화를 하면서 while문을 생략할 수도 있다는 것이다.

```
device = 0xAABB;  
while (device != 0xAABB) {  
    ...  
}
```

인터럽트 핸들러 사용할 때

- 인터럽트 핸들러인 port_read()
 - 인터럽트가 발생해서 port_read()가 호출 될 때, PORTA, PORTB 값을 읽어 배열에 저장하며,
 - main에서는 저장된 값이 바뀌면 알람이 울리도록 한다.
- 하지만 똑똑하지만 (일하기 싫은) 컴파일러는 인터럽트 핸들러에 의해 값이 바뀔 수 있다는 사실 자체를 모르며, 그저 if를 없애버린다는 것
→ volatile uint8_t input_value[2] 로 하면 된다.
- volatile 을 붙이면,
컴파일러는 언제든지 이 값이 바뀔 수 있다고 인지하게 된다.

```
uint8_t input_value[2] = { 0x0, 0x0 };

void interrupt port_read(void) {
    input_value[0] = PORTA;
    input_value[1] = PORTB;
}

int main() {

    if (input_value[0] != input_value[1]) {
        //알람을 울린다.
    }

    return 0;
}
```

- 메모리 주소를 가진 I/O 레지스터
- 인터럽트 핸들러가 값을 변경하는 전역 변수
- 등등 최적화에 의해 오류가 발생할 가능성이 있는 변수에 쓰인다.

너무 남발하면, 오히려 코드 최적화를 할 수 없으므로 적재적소에 volatile을 붙인다.

main argument

C언어의 `main()` 는 실행 파일 옵션을 매개 변수로 받을 수 있다.

- `argc` : `main()` 에 전달되는 정보의 개수
- `argv[]` : `main()` 에 전달되는 정보, 첫 번째 문자열은 실행 경로로 고정

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    printf("argc = %d\n", argc);
    printf("argv = %s\n", argv[0]);

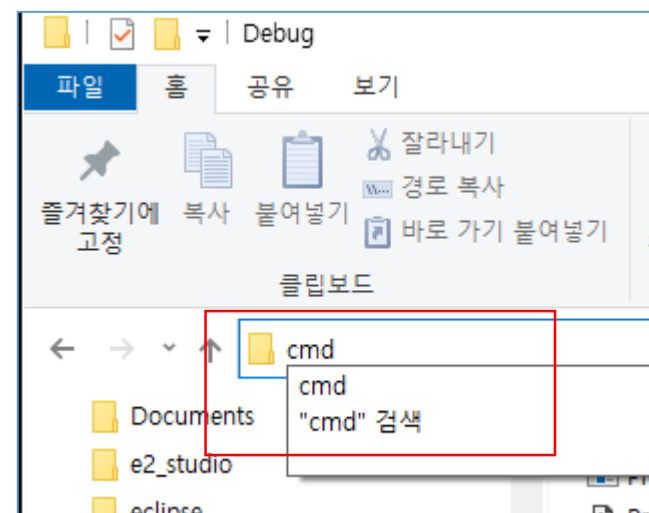
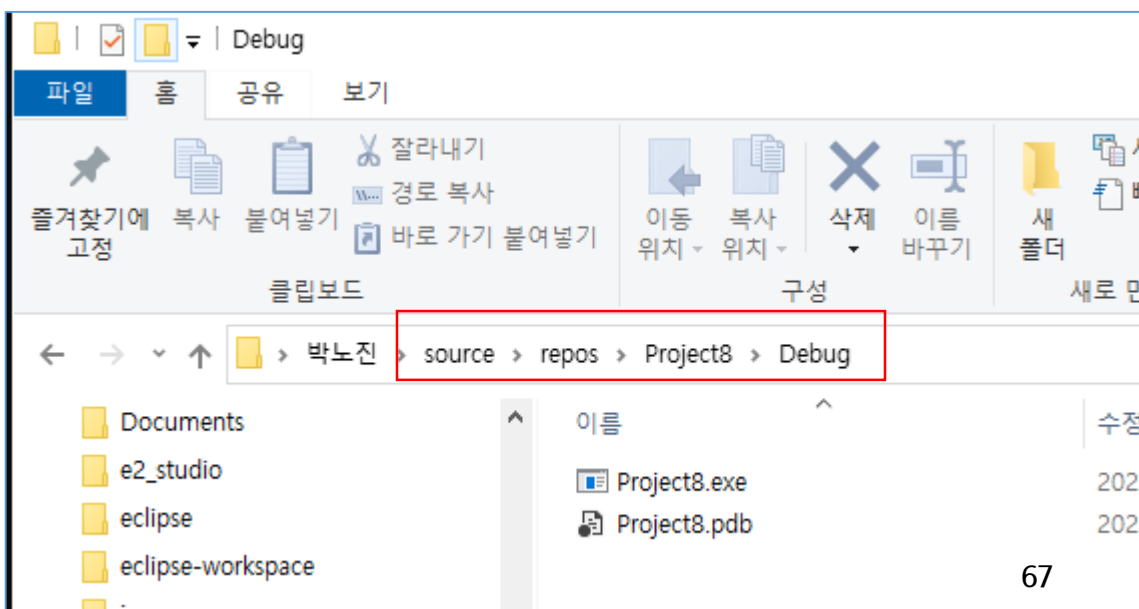
    return 0;
}
```

<https://gist.github.com/hoconoco/e351854b709ae5c538ee3fbef075ed24>

콘솔 창에 든 프로그램의 실행 경로로 들어간다.

- 해당 디렉토리에서 디렉토리 경로 창에 cmd를 입력하고 Enter 키를 누른다.
- 해당 디렉토리로 시작하는 콘솔 창이 열린다.

```
C:\> 선택 Microsoft Visual Studio 디버그 콘솔  
argc = 1  
argv = C:\Users\nojin\source\repos\Project8\Debug\Project8.exe
```



실행 파일의 이름을 입력하면 실행 된다.

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19043.2006]
(c) Microsoft Corporation. All rights reserved.

C:\Users\nojin\source\repos\Project8\Debug>Project8.exe
argc = 1
argv = Project8.exe

C:\Users\nojin\source\repos\Project8\Debug>
```

main argument 를 이용해
프로그램을 실행하면서 값을 넣는다.

```
C:\Users\hojin\source\repos\Project8\Debug>Project8.exe hi hello 1234
argc = 4
argv[0] = Project8.exe
argv[1] = hi
argv[2] = hello
argv[3] = 1234
```

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

    printf("argc = %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }

    return 0;
}
```

<https://gist.github.com/hoconoco/1b32a0324350bf3f291b38574dc17833>

디렉토리를 생성한 뒤 테스트한다.

- `mkdir ~/test`
- `cd ~/test`
- `vi argu.c`
- 코드 복사 붙여넣기
- `gcc ./argu.c`
- `./a.out hifaker hello 100`

```
ssafy@ssafy-VirtualBox:~$ mkdir test
ssafy@ssafy-VirtualBox:~$ cd ./test
ssafy@ssafy-VirtualBox:~/test$ vi argu.c
ssafy@ssafy-VirtualBox:~/test$ ls
argu.c
ssafy@ssafy-VirtualBox:~/test$ gcc ./argu.c
ssafy@ssafy-VirtualBox:~/test$ ls
a.out  argu.c
ssafy@ssafy-VirtualBox:~/test$ ./a.out hifaker hello 100
argc = 4
argv[0] = ./a.out
argv[1] = hifaker
argv[2] = hello
argv[3] = 100
ssafy@ssafy-VirtualBox:~/test$
```

Day6-3. SSD Test Shell 제작 프로젝트

챕터의 포인트

- 가상 SSD 제작
- Test Shell Application
- Test Script 작성하기

가상 SSD 제작

SSD 제품을 테스트 할 수 있는 Test Shell 을 제작

1. SSD 를 가상으로 프로그래밍으로 구현한다.
2. Test Shell 프로그램을 제작하여 SSD 동작을 테스트 할 수 있다.
3. 다양한 Test Script를 제작한다.



Test 수행



1. SSD

- H/W 를 S/W로 구현한다.

2. Test Shell Application

- 테스트 프로그램

3. Test Script

- 테스트 프로그램 內 Test Code API()



ssd.c



TestShell.c

최종 제작해야하는 소스코드는 두 가지

- ssd.c
- testshell.c

1. 어떤 환경에서 개발을 하든 상관이 없다.

- Window / Linux / AWS EC2 ...

2. SSD를 개발하고 TestShell 로 SSD에 신호를 보내야 하는 데,
어떤 방식을 사용하든 상관이 없다.

- 깐부와 함께 해결한다.

자소서 or 면접에 가서 말할 수 있을 정도의 프로젝트이므로,
실제와 유사하게 만들 수록 더 말할 거리가 풍부할 것이다.

저장할 수 있는 공간

- 저장할 수 있는 최소 공간의 사이즈는 4KB
(한 글자 = 약 1 Byte 으로 간주했을 때, 4,096 글자 저장 가능 공간)
- 각 공간마다 LBA (Logical Block Address) 라는 주소를 가짐
- SSD 는 OS로부터
Read / Write / Unmap 등
- 다양한 명령어를 전달받는다.



최소화된 기능 수행

- Read 명령어와 Write 명령어만 존재
 - 실제 SSD에 쓰이는 명령어를 추가로 더 구현해도 좋다!
- LBA 단위는 4 Byte (실제로는 4KB 이지만, 우리가 만들 최소 저장공간 사이즈는 4 Byte)
- LBA 0 ~ 99 까지 100 칸을 저장할 수 있다.

총 400 Byte를 저장 할 수 있는 가상 SSD 를 구현



ssd.c

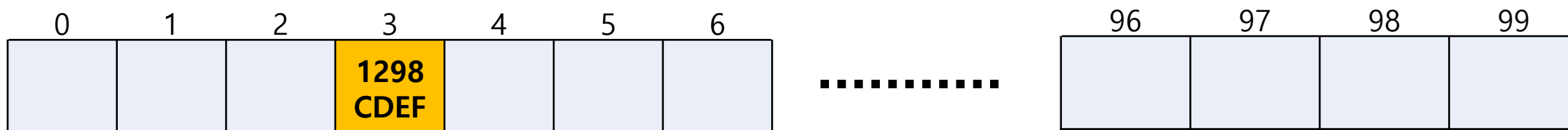


만들어야 하는 가상 SSD 프로그램의 이름

- “**ssd**”

Write 명령어 사용 예시 1

- **ssd W 3 0x1298CDEF** : 3 번 LBA 영역에 값 0x1298CDEF 를 저장한다.

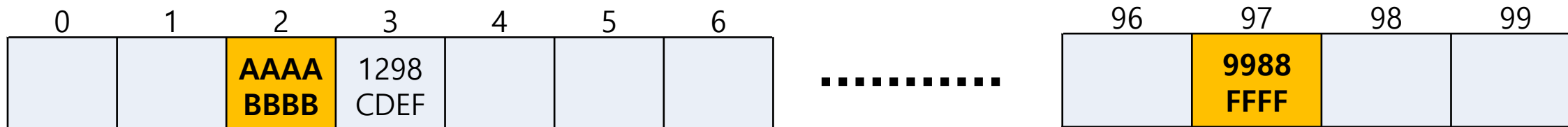


Write 명령어 사용 예시 2

- `ssd W 2 0xAAAABBBB`
- `ssd W 97 0x9988FFFF`

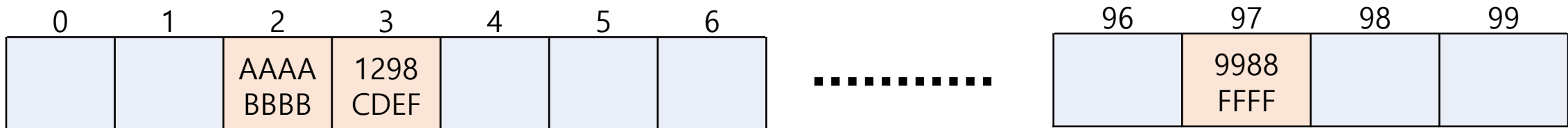
출력결과 : 없음 (저장만 수행)

- `ssd W 3 0x1234ABCD` 를 입력해도 출력으로 나타나지 않는다.
- `0x1234ABCD` 값은 `nand.txt` 에 저장된다.



nand.txt 파일

- nand.txt 를 생성해야 한다.
- 사용자가 Write 할 때 마다, SSD 내부 (Nand) 에 기록이 된다.
- 이를 모사하여, nand.txt 파일에 값을 저장 해 둔다.



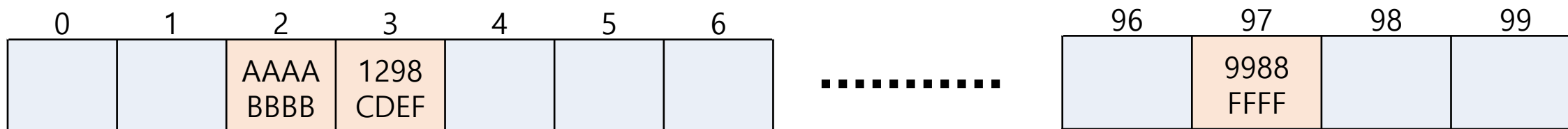
Read 명령어 사용 예시

- `ssd R 2`
→ 읽혀진 결과 : 0xAAAABBBB
- `ssd R 97`
→ 읽혀진 결과 : 0x9988FFFF

Write 와 마찬가지로

Read 명령어를 수행해도 화면에 결과를 출력하는 것이 아니다!

- `result.txt` 에 Read 의 결과를 출력한다.



화면 출력하지 않고, result.txt 파일에 결과를 저장

- Write 명령어 수행 시

- result.txt 파일 건드리지 않음
- Write는 내부적으로 기록만 수행한다.

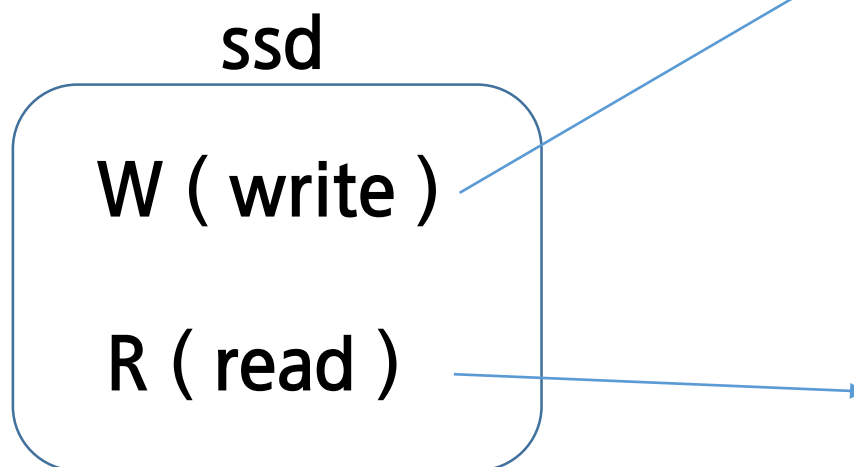
- Read 명령어 수행 시

- result.txt 파일 내용이 교체 된다.
- result.txt 에는 Read 를 한 마지막 결과만 저장된다.

Write 명령어 사용시 저장되는 장소
nand.txt (ssd의 nand 메모리)



Read 명령어 사용시 출력되는 장소
result.txt



Read 명령 시 결과 값이 result.txt 파일에 저장

- 아래 표와 같이 명령어를 위에서부터 입력하면 최종적으로 0xFF1100AA 가 result.txt 에 저장되어 있다.

Write 명령 시 nand.txt 파일에 값을 저장

- nand.txt 에 Data 가 저장되는 방식은 간부와 함께 해결한다.

입력	출력(result.txt)
ssd W 20 0x1289CDEF	-
ssd R 20	0x1289CDEF
ssd R 19	0x00000000
ssd W 10 0xFF1100AA	
ssd R 10	0xFF1100AA

데이터 범위

- LBA : 0 ~ 99, 10진수
- 값 : **항상 0x가 붙으며 10 글자로 표기한다.** (0x00000000 ~ 0xFFFFFFFF)

Read 명령어

- ssd R [LBA]
- result.txt 에 Read한 결과 값이 적힌다. (**기존 데이터는 사라진다.**)
- 한번도 안 적은 곳은 **0x00000000** 으로 읽힌다.

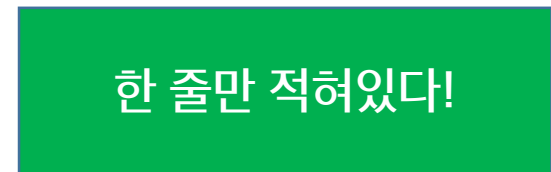
Write 명령어 사용시 저장되는 장소
nand.txt (ssd의 nand 메모리)



Write 명령어

- ssd W [LBA] [값]
- nand.txt 에 저장한 값이 기록된다.
 - 기록 되는 형식은 각 팀에서 알아서 해결한다.

Read 명령어 사용시 출력되는 장소
result.txt

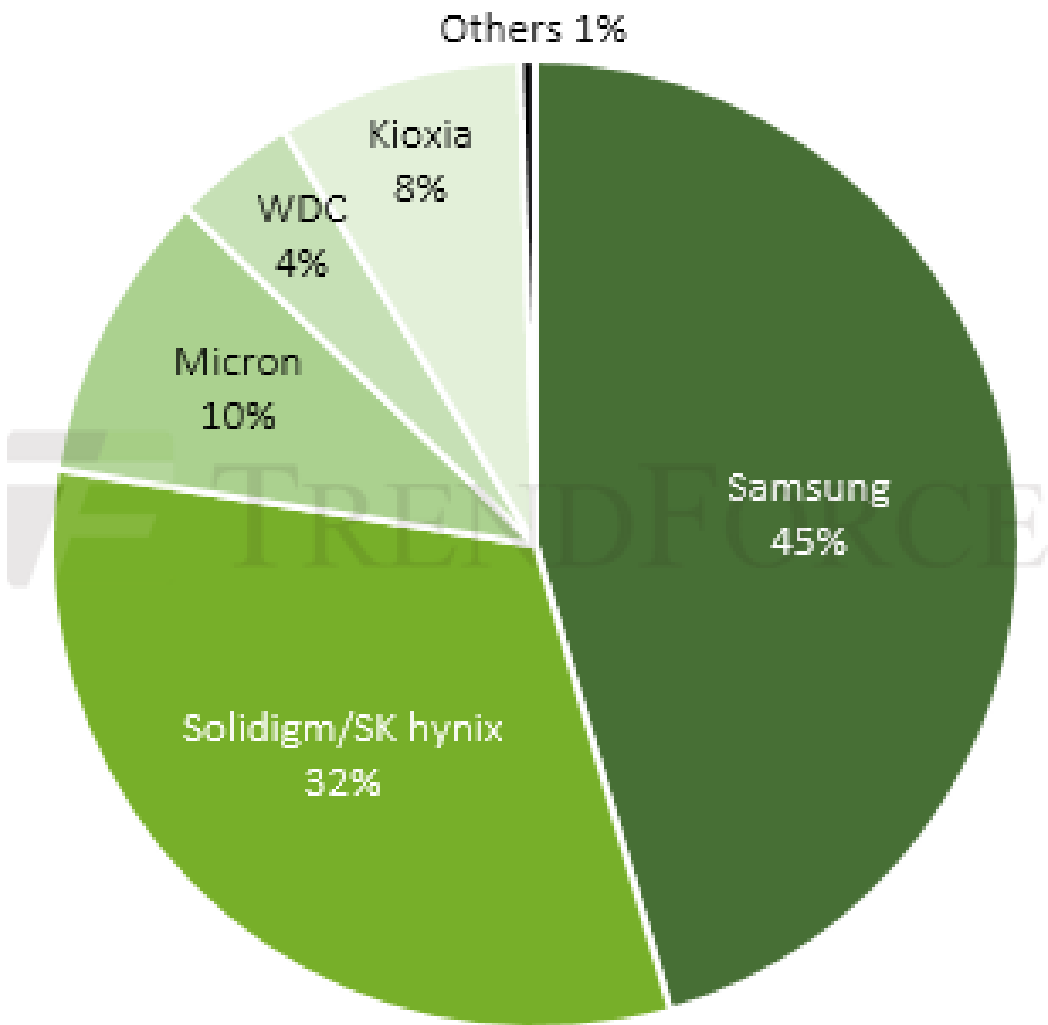


2023년 4분기

B2B PC 용 저장장치 (SSD) 시장 점유율

- 세계 1위 : 삼성전자
- 세계 2위 : Solidigm(솔리다임) /SK Hynix

Market Shares of Enterprise SSD Suppliers, 4Q23

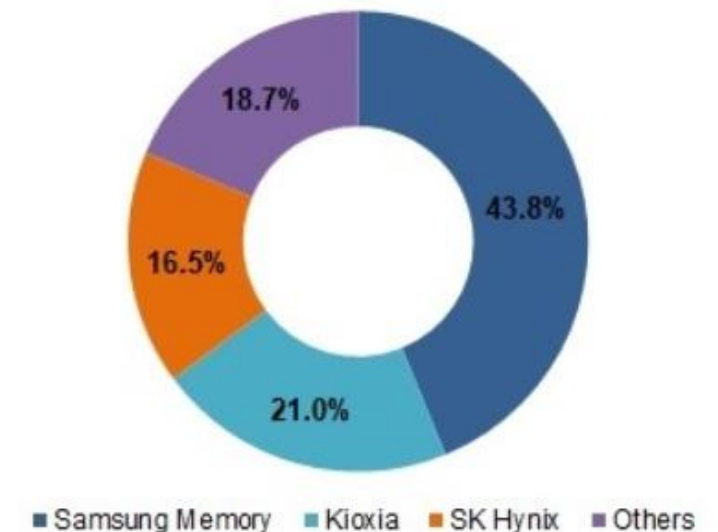


Source: TrendForce, Apr., 2024

Mobile용 저장장치 (emmc / ufs)

- 삼성전자 : 세계 1위
- SKHynix 세계 4위
- eMMC
 - embedded Multi-Media Card
 - 데이터 고속처리를 위해 모바일 기기에 내장하는 메모리 반도체
- UFS
 - Univesal Flash Storage
 - 차세대 초고속 플래시 메모리, eMMC 보다 2.7배 빠른 속도 가능 /

Smartphone NAND Flash Market Revenue Share Q1 2020

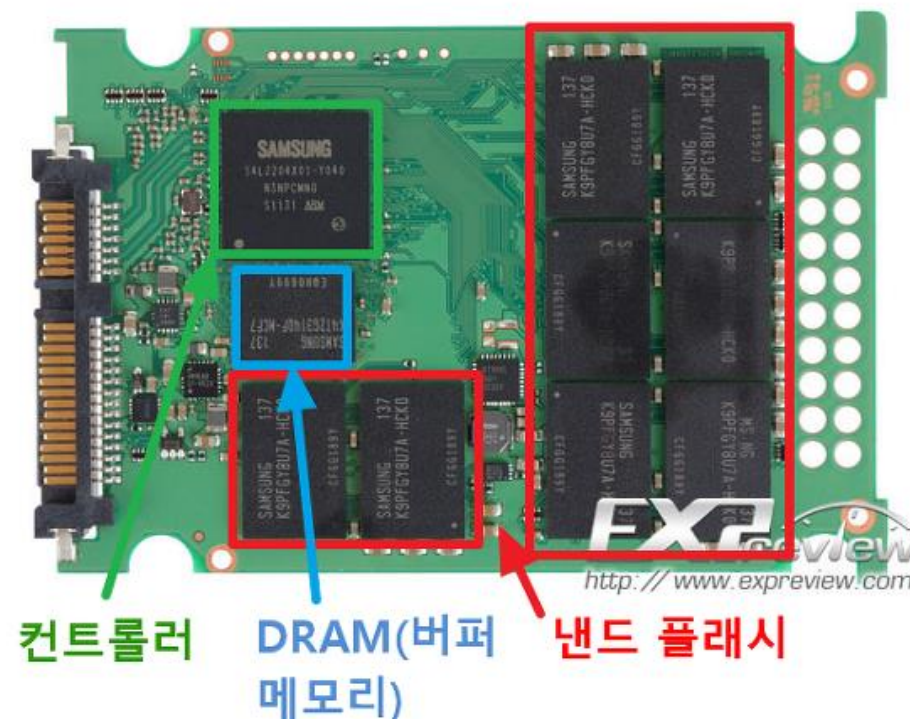


컨트롤러 역할

- OS 는 Filesystem를 거쳐 LBA 주소에 특정 값 R/W 요청
- 컨트롤러가 이 명령어들을 받아 Nand Flash Memory 에 저장
- 우리가 만드는 SSD 프로그램의 역할이다.

NAND Flash Memory

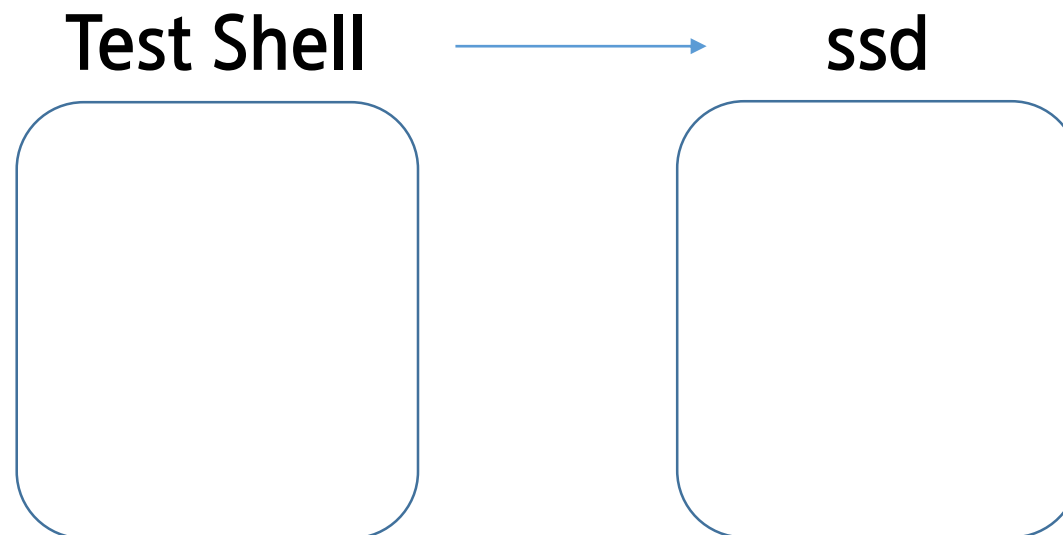
- 실제로 값들이 저장되는 곳
- 컨트롤러에 의해 값이 저장이 된다.
- 우리 프로젝트에서 nand.txt 에 해당한다.



Test Shell Application

SSD를 테스트 할 수 있는 프로그램

- Shell 이 동작하여 사용자 입력을 받는다.
- 사용 가능 명령어
 - write
 - read
 - exit
 - help
 - fullwrite
 - fullread



read / write 명령어 수행시,
제작한 "ssd" app을 실행시켜 값 읽기 / 저장 명령을 수행한다.

write 3 0xAAAABBBB

- 3번 LBA 에 0xAAAABBBB 를 기록한다.
- ssd 에 명령어를 전달한다.

read 3

- 3번 LBA 를 읽는다.
- ssd 에 명령어를 전달한다.
- **result.txt 에 적힌 결과를 화면에 출력한다.**

exit 명령어

- Shell 이 종료된다.

help 명령어

- 각 명령어당 사용 방법을 출력한다.

fullwrite 명령어

- LBA 0 번부터 99 번 까지 Write를 수행한다.
- **ssd 전체에 값이 써진다.**
- ex) fullwrite 0xABCDFFFF
→ 모든 LBA에 값 0xABCDFFF 가 적힌다.

fullread 명령어

- LBA 0 번부터 99 번 까지 Read를 수행한다.
- **ssd 전체 값을 모두 화면에 출력한다.**
- ex) fullread
→ 모든 LBA의 값들이 화면에 출력된다.

기능 구현 시 유의사항

- 입력 받은 매개변수가 유효성 검사 수행
 - 파라미터의 Format이 정확해야 함
 - LBA 범위는 0 ~ 99
 - A ~ F, 0 ~ 9 까지 숫자 범위만 허용
- 없는 명령어를 수행하는 경우 **"INVALID COMMAND"** 을 출력
 - 어떠한 명령어를 입력하더라도 **segment fault**가 나오면 안된다.

Test Script 작성하기

실제 Test를 수행하는 함수를 제작

- Test 를 수행하는 명령어의 집합을 Test Script라고 한다.
- 주로 “검증”팀에서 Test Script를 제작한다.
- TestShell 내부에서 동작하는 API(함수)이다.



Test Script

TestShell.c

TestApp1 개요

- Test Shell 에서 “testapp1” 명령어를 입력하면 Script가 수행된다.
- 먼저 fullwrite를 수행한다.
- fullread를 하면서, write 한 값 대로 read가 되는지 확인한다.
→SSD 정상 동작하는지 확인하는 프로그램

Full Write 후 Read Compare 수행

입력 형식

>>testapp1

Write Aging 후 Read Compare

- 0 ~ 5 번 LBA 에 0xAAAABBBB 값으로 총 30번 Write를 수행한다.
- 0 ~ 5 번 LBA 에 0x12345678 값으로 1 회 Over Write를 수행한다.
- 0 ~ 5 번 LBA Read 했을 때 정상적으로 값이 읽히는지 확인한다.

입력 형식

>>testapp2

내일 방송에서 만나요!

삼성 청년 SW 아카데미

[과제1] p.9 코드와 결과물 캡처해서 제출

[과제2] p.15~17 코드와 결과물 캡처해서 제출

[과제3] p.22 코드와 결과물 캡처해서 제출

[과제4] p.29 코드와 결과물 캡처해서 제출

[과제4] p.30~p.42 코드와 결과물 캡처해서 제출

[과제5] p.50~p.61 실습 진행 후 결과 캡처해서 제출

[과제6] p.65~p.70 실습 진행 후 결과 캡처해서 제출

[과제7] SSD Project 결과물 캡처해서 제출