
Introduction to Software Design

C04. Array, Pointer

Tae Hyun Kim

Topics Covered

- 배열(Array)
- 문자열(String)
- 포인터(Pointer)
- 배열과 포인터의 유사점/차이점
- 포인터의 증가/감소 연산
- 함수의 인자로 배열을 전달하기

배열(Array)

- 동일한 자료형의 변수를 여러 개 모아놓은 것
- ex) 100명의 나이를 저장하는 경우를 생각해보자
 - 만일 이렇게 한다면?
 - `int age1, age2, age3, ... , age99, age100;`
 - 평균을 구하고 싶다면? 최대값을 구하고 싶다면?
 - 매우 불편할 것임!
- `int ages[100];` // 배열
- 반복문을 통해 각각의 요소에 접근 가능

배열의 선언 및 배열 요소의 접근

```
int oneDimArr [4];
```

int 배열을 이루는 요소(변수)의 자료형
oneDimArr 배열의 이름
[4] 배열의 길이



생성되는 배열의 형태

- 배열 ages의 첫 번째 요소: ages[0]
- 배열 ages의 두 번째 요소: ages[1]
- ...
- 배열 ages의 i+1 번째 요소: ages[i]

array[i] → subscript operator
혹은 indexing operator라고
불림

- 배열 요소의 위치 정보를 나타내는 인덱스(index)는 1이 아닌
0부터 시작!

배열에서 []의 사용법

- 배열을 선언할 때

```
int ages[5];
```

– []안의 숫자는 배열의 길이를 의미

- 배열의 요소에 접근할 때

```
num = ages[2];  
ages[1] = 30;
```

– []안의 숫자(i)는 각 요소가 배열에서 몇 번째(i+1)에 있는지를 의미

배열을 선언과 동시에 초기화하기

초기화리스트

```
int arr1[5]={1, 2, 3, 4, 5};
```



```
int arr2[ ]={1, 2, 3, 4, 5, 6, 7};
```



```
int arr2[7]={1, 2, 3, 4, 5, 6, 7};
```

컴파일러가 배열의 길이정보 채움

배열의 길이 구하기

- `int arr[5];`
- `sizeof(arr) ?`
- $\rightarrow 20$: 전체 바이트 크기
- 배열의 길이를 구하려면: `sizeof(arr) / sizeof(int)`
- $\rightarrow 5$

C & Python Examples

- C

```
#include <stdio.h>
int main()
{
    int arr[] = {2,4,6,8,10};
    printf("len: %ld\n", sizeof(arr)/sizeof(int));

    int sum = 0;
    for(int i=0; i<sizeof(arr)/sizeof(int); ++i)
        printf("%d\n", arr[i]);

    int min = arr[0];
    int max = arr[0];
    for(int i=0; i<sizeof(arr)/sizeof(int); i++)
    {
        if(arr[i] < min)
            min = arr[i];
        if(arr[i] > max)
            max = arr[i];
    }
    printf("min: %d\n", min);
    printf("max: %d\n", max);
    return 0;
}
```

- Python

```
arr = [2,4,6,8,10]
print('len: %ld'%len(arr))

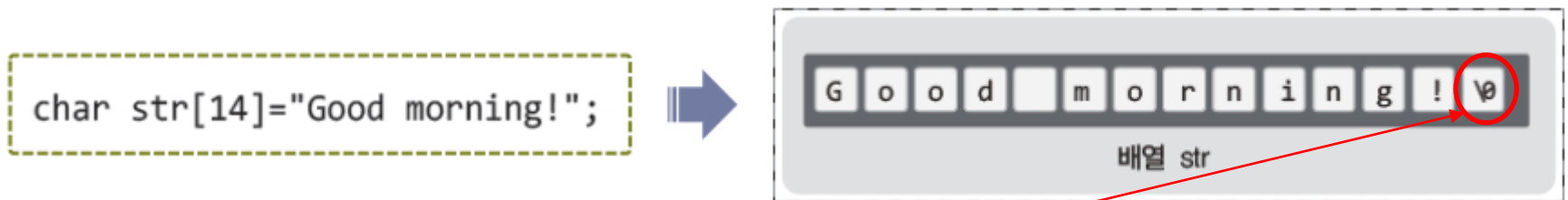
sum = 0
for i in range(len(arr)):
    print('%d'%arr[i])

print('min: %d'%min(arr))
print('max: %d'%max(arr))
```

- Python에서 C의 array와 비슷한 용도로 일반적으로 쓰이는 것은 list이다. (이 두 개가 정확히 같지는 않다)

문자열(String)

- “Hello”
- C언어에서는 문자열도 배열로 표현된다
- -> char형 배열!



널(null) 문자 (\0): C에서 문자열의 끝을 표시하기 위해 꼭 필요하다!

13글자로 된 문자열을 저장하기 위해서는 길이가 14이상인 배열이 필요하다!

C와 Python의 문자열

- 문자열을 사용하려면 메모리 공간에 저장된 문자열 데이터의 시작 주소와 끝 주소를 알아야 한다.
- C의 문자열
 - 배열이며, 배열은 본래 메모리 공간상의 시작 주소로 접근한다.
 - 문자열 맨 끝 다음에 오는 null 문자로 끝 주소를 알 수 있다.
 - 그래서 C의 문자열을 null-terminated string이라고 부르기도 함.
- Python의 문자열
 - (엄밀히 말하면 python interpreter의 구현에 따라 다르지만, 보통 많이 사용하는 CPython의 경우에는,)
 - `str` 객체이며, 메모리 공간상의 시작 주소와 문자열의 길이를 가지고 있다.
 - 시작 주소와 길이를 통해 끝 주소를 알 수 있다.

배열과 문자열의 차이점

```
char arr1[ ] = {'H', 'i', '~'};
```

```
char arr2[ ] = {'H', 'i', '~', '\0'}; == char arr2[] = "Hi~";
```

arr1은 문자열이 아닌 문자 배열, 반면 arr2는 문자열!
널 문자의 존재여부는 문자열의 판단여부가 된다.

C & Python Examples

- C

```
#include <stdio.h>

int main()
{
    char str[] = "Good morning!";

    printf("배열 str의 길이: %d\n",
sizeof(str)/sizeof(char));
    printf("널문자를 문자로 출력: %c\n", str[13]);
    printf("널문자를 숫자로 출력: %d\n", str[13]);

    printf("%s\n", str);

    str[8] = '\0';
    printf("%s\n", str);

    str[6] = '\0';
    printf("%s\n", str);

    str[1] = '\0';
    printf("%s\n", str);

    return 0;
}
```

- Python

```
str = 'Good morning!'
str[0] = '\0'
# TypeError: 'str'
object does not support
item assignment
```

- Python의 string은
변경불가능(immutable)
하므로 이러한 연산이
불가능하다.

scanf()를 이용한 문자열 입력

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

```
int main()  
{
```

```
    char str[50];
```

```
    printf("문자열 입력: ");
```

```
    scanf("%s", str);
```

```
    printf("입력받은 문자열: %s\n", str);
```

```
    return 0;
```

```
}
```

배열 이름 str 앞에는 &를
붙이지 않는다!

만일 he is my friend라고 입력하면?

-> scanf는 공백을 기준으로
데이터를 구분하므로 공백을
포함하는 문자열을 읽어 들이지
못한다.

메모리 레이아웃

- 일직선으로 늘어선 1차원 배열로 생각하면 됨.
 - 0번지 ~ 최대번지
 - 번지수는 1바이트마다 1씩 증가
 - 예:

메모리 주소(번지수)

해당 번지에 저장된 내용

| | | | | | | | | | | | | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 10241 | 10242 | 10243 | 10244 | 10245 | 10246 | 10247 | 10248 | 10249 | 10250 | 10251 | 10252 | 10253 | 10254 | 10255 | 10256 | 10257 | 10258 | 10259 | 10260 |
| | | | | | | | | | | | | | | | | | | | |
| 10261 | 10262 | 10263 | 10264 | 10265 | 10266 | 10267 | 10268 | 10269 | 10270 | 10271 | 10272 | 10273 | 10274 | 10275 | 10276 | 10277 | 10278 | 10279 | 10280 |
| | | | | | | | | | | | | | | | | | | | |

(위 그림의 주소 숫자는 하나의 예에 불과함...)

int형 변수가 저장된 모습

```
int num1 = 5;  
int num2 = 129;
```

00000000 00000000 00000000 00000101

| | | | | | | | | | | | | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 10241 | 10242 | 10243 | 10244 | 10245 | 10246 | 10247 | 10248 | 10249 | 10250 | 10251 | 10252 | 10253 | 10254 | 10255 | 10256 | 10257 | 10258 | 10259 | 10260 |
| | | | | | num1 | | | | | | | | | | | | | | |
| 10261 | 10262 | 10263 | 10264 | 10265 | 10266 | 10267 | 10268 | 10269 | 10270 | 10271 | 10272 | 10273 | 10274 | 10275 | 10276 | 10277 | 10278 | 10279 | 10280 |
| | | | | | | | | | | | num2 | | | | | | | | |

00000000 00000000 00000000 10000001

address-of 연산자: 주소 값을 반환

&num1 == ? -> 10246

&num2 == ? -> 10272

(참고)

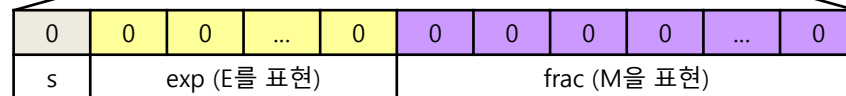
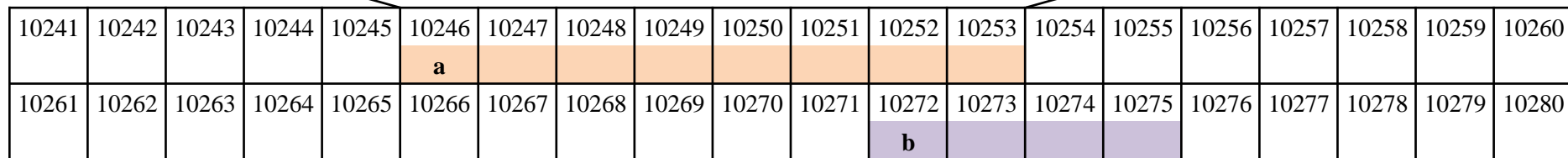
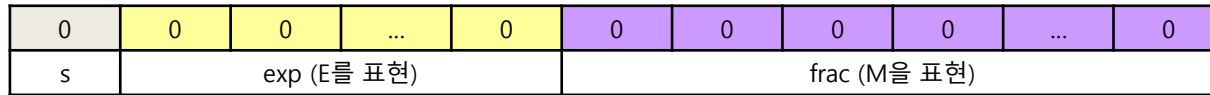
ARM 계열 – Big-endian : 위에 표시된 순서

Intel x86 계열 – Little-endian : 바이트 단위로 반대 순서

예) 5 -> 00000101 00000000 00000000 00000000

double, float형 변수가 저장된 모습

```
double a = 3.14;  
float b = 1.1;
```



&a == ? -> 10246

&b == ? -> 10272

char형 변수, 문자열이 저장된 모습

```
char ch = 'A';  
char str[10] = "Hello";
```

01000001 ('A'==65)

| | | | | | | | | | | | | | | | | | | | |
|-------|-------|-------|--------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 10241 | 10242 | 10243 | 10244 | 10245 | 10246 | 10247 | 10248 | 10249 | 10250 | 10251 | 10252 | 10253 | 10254 | 10255 | 10256 | 10257 | 10258 | 10259 | 10260 |
| | | | ch 'A' | | | | | | | | | | | | | | | | |
| 10261 | 10262 | 10263 | 10264 | 10265 | 10266 | 10267 | 10268 | 10269 | 10270 | 10271 | 10272 | 10273 | 10274 | 10275 | 10276 | 10277 | 10278 | 10279 | 10280 |
| | | | | | 'H' | 'e' | 'l' | 'l' | 'o' | '\0' | | | | | | | | | |

&ch == ? -> 10244

str == ? -> 10266

포인터(Pointer)

- 변수의 메모리 주소를 저장하는 변수

```
double a = 3.14;  
float b = 1.1;
```

| | | | | | | | | | | | | | | | | | | | |
|-------|-------|-------|-------|-------|------------|-------|-------|-------|-------|-------|------------|-------|-------|-------|-------|-------|-------|-------|-------|
| 10241 | 10242 | 10243 | 10244 | 10245 | 10246 a | 10247 | 10248 | 10249 | 10250 | 10251 | 10252 | 10253 | 10254 | 10255 | 10256 | 10257 | 10258 | 10259 | 10260 |
| 10261 | 10262 | 10263 | 10264 | 10265 | 10266 | 10267 | 10268 | 10269 | 10270 | 10271 | 10272 b | 10273 | 10274 | 10275 | 10276 | 10277 | 10278 | 10279 | 10280 |

&a == ? -> 10246

&b == ? -> 10272

```
int a_address = &a;
```

```
// 변수 a_address에 10246 저장??
```

포인터(Pointer)

| | | | | | | | | | | | | | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 10241 | 10242 | 10243 | 10244 | 10245 | 10246 | 10247 | 10248 | 10249 | 10250 | 10251 | 10252 | 10253 | 10254 | 10255 | 10256 | 10257 | 10258 | 10259 | 10260 |
| | | | | | a | | | | | | | | | | | | | | |
| 10261 | 10262 | 10263 | 10264 | 10265 | 10266 | 10267 | 10268 | 10269 | 10270 | 10271 | 10272 | 10273 | 10274 | 10275 | 10276 | 10277 | 10278 | 10279 | 10280 |
| | | | | | | | | | | | b | | | | | | | | |

- 정확한 방법:

```
double* a_address = &a;
```

- 다양한 자료형의 변수의 주소를 저장하기 위한
포인터 형이 따로 존재
 - int*
 - double*
 - ...

포인터 : 변수의 메모리 주소를 저장하는 변수

- `int*` : int형 포인터 – int형 변수의 주소를 저장
- `int* pnum1;` // int형 포인터 변수 `pnum1`
- `double*` : double형 포인터 – double형 변수의 주소를 저장
- `double* pnum2;` // double형 포인터 변수 `pnum2`
- `char*, float*, ...`

포인터 변수가 메모리에 저장된 모습

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

```
int main()
```

```
{
```

```
    char ch1 = 'a';
```

```
    char* pch1 = &ch1;
```

```
    printf("value of ch1: %d\n", ch1);
```

```
    printf("address of ch1: %p\n", &ch1);
```

```
    printf("value of pch1: %p\n", pch1);
```

```
    printf("address of pch1: %p\n", &pch1);
```

```
    return 0;
```

```
}
```

value of ch1: 97
address of ch1: 1636819
value of pch1: 1636819
address of pch1: 1636804

실제로 할당되는 메모리 주소
는 실행할 때마다 달라진다.

%p로 메모리 주소를 출력하면
16진수로 출력된다.

하지만 본 강의 슬라이드에서
는 편의상 메모리 주소를 10진
수로 표시하기로 한다.

포인터 변수가 메모리에 저장된 모습

```
value of ch1: 97  
address of ch1: 1636819  
value of pch1: 1636819  
address of pch1: 1636804
```

| | | | | | | | | | |
|---------|---------|---------|--------------|---------|---------|---------|---------|---------|---------|
| 1636801 | 1636802 | 1636803 | 1636804 | 1636805 | 1636806 | 1636807 | 1636808 | 1636809 | 1636810 |
| | | | pch1 1636819 | | | | | | |
| 1636811 | 1636812 | 1636813 | 1636814 | 1636815 | 1636816 | 1636817 | 1636818 | 1636819 | 1636820 |
| | | | | | | | | ch1 'a' | |

points to
(가리키다)

- 그래서 변수의 메모리 주소를 저장하는 변수를 **pointer** 라고 한다.

포인터에서 *의 사용법

- 포인터 변수를 선언할 때

```
int* pnum;
```

- 포인터가 가리키는 변수를 참조할 때 (간접참조연산자)

```
num2 = *pnum;  
*pnum = 20;
```

참고1) *의 띄어쓰기는 상관없다

```
int* pnum;    // (1)
```

```
int * pnum;    // (2)
```

```
int *pnum;    // (3)
```

int*를 하나의 자료형으로 인식하기 쉬운 방법인 **int**와 *를 붙여쓰는 (1)번을 추천.

참고2) 아래와 같은 선언도 가능
int num, * pnum;
하지만 **int***를 붙여 쓰지 않기 때문에 추천하지 않음.

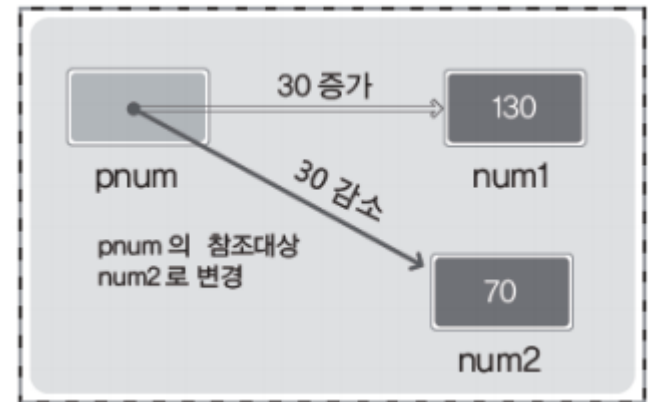
포인터가 가리키는 변수가 바뀔 수도 있다

```
int main(void)
{
    int num1=100, num2=100;
    int * pnum;

    pnum=&num1;    // 포인터 pnum이 num1을 가리킴
    (*pnum)+=30;    // num1+=30; 과 동일

    pnum=&num2;    // 포인터 pnum이 num2를 가리킴
    (*pnum)-=30;    // num2-=30; 과 동일

    printf("num1:%d, num2:%d \n", num1, num2);
    return 0;
}
```



다양한 포인터 형이 존재하는 이유

- 어차피 어떤 변수가 저장된 메모리 공간의 시작 주소만 알면 되지 않나요? (X)

int형 포인터 변수로 * 연산을 통해 메모리(변수) 접근 시

4바이트 메모리 공간에 부호 있는 정수의 형태로 데이터를 읽고 쓴다.

double형 포인터 변수로 * 연산을 통해 메모리(변수) 접근 시

8바이트 메모리 공간에 부호 있는 실수의 형태로 데이터를 읽고 쓴다.

```
int main(void)
{
    double num=3.14;
    int * pnum=&num;
    printf("%d", *pnum);
    . . . .
}
```

형 불일치! 컴파일은 된다.
pnum이 가리키는 것은 double형 변수인데, pnum이 int형 포인터 변수이므로 int형 데이터처럼 해석!

잘못된 포인터의 사용

```
int main(void)
{
    int * ptr;
    *ptr=200;
    . . . .
}
```

ptr이 쓰레기 값으로 초기화 된다. 따라서 200이 저장되는 위치는 어디인지 알 수 없다! 매우 위험한 행동!

```
int main(void)
{
    int * ptr=125;
    *ptr=10;
    . . . .
}
```

포인터 변수에 125를 저장했는데 이곳이 어디인가? 역시 매우 위험한 행동!

널(Null) 포인터

```
int main(void)
{
    int * ptr1=0;
    int * ptr2=NULL;
    . . . .
}
```

잘못된 포인터 연산을 막기 위해서 특정한 값으로 초기화하지 않는 경우에는 **널 포인터**로 초기화하는 것이 안전하다.

널 포인터 NULL은 숫자 **0**을 의미한다. 그리고 **0**은 **0번지**를 뜻하는 것이 아니라, 아무것도 가리키지 않는다는 의미로 해석이 된다.

보통 아래와 같이 의미 있는 주소값이 포인터에 대입되어 있는지 체크한 후 필요한 작업을 하도록 하는 식으로 코드 작성.

```
if (ptr2 == NULL)
    ptr2 = &num1;

*ptr2 = 10;
```

32bit & 64bit 아키텍처

- 32bit CPU

- 32bit OS

- 메모리 주소를 32bit 로 표현

- 시스템이 인식할 수 있는 최대 메모리

- → 2^{32} 바이트 = 4GB (이론상)

- 포인터의 크기는?

- → 32bit = 4 바이트

- 64bit CPU

- 64bit OS

- 메모리 주소를 64bit 로 표현

- 시스템이 인식할 수 있는 최대 메모리

- → 2^{64} 바이트 = 16TB (이론상)

- 포인터의 크기는?

- → 64bit = 8 바이트

배열의 이름

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

```
int main()  
{
```

```
    int arr[3] = {5, 10, 20};
```

```
    printf("배열 이름 자체의 값: %p\n", arr);
```

```
    printf("첫 번째 요소의 주소: %p\n", &arr[0]);
```

```
    printf("두 번째 요소의 주소: %p\n", &arr[1]);
```

```
    printf("세 번째 요소의 주소: %p\n", &arr[2]);
```

```
    return 0;
```

```
}
```

| | |
|--------------|---------|
| 배열 이름 자체의 값: | 1638052 |
| 첫 번째 요소의 주소: | 1638052 |
| 두 번째 요소의 주소: | 1638056 |
| 세 번째 요소의 주소: | 1638060 |

배열이 메모리에 저장된 모습

```
int arr[3] = {5, 10, 20};
```

| | |
|--------------|---------|
| 배열 이름 자체의 값: | 1638052 |
| 첫 번째 요소의 주소: | 1638052 |
| 두 번째 요소의 주소: | 1638056 |
| 세 번째 요소의 주소: | 1638060 |

arr의 값 = arr[0]의 주소

주소값의 차이가 4
: int형 배열이기 때문

| | | | | | | | | | | | | | | | |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1638050 | 1638051 | 1638052 | 1638053 | 1638054 | 1638055 | 1638056 | 1638057 | 1638058 | 1638059 | 1638060 | 1638061 | 1638062 | 1638063 | 1638064 | 1638065 |
| | | arr[0] | 5 | | | arr[1] | 10 | | | arr[2] | 20 | | | | |

- 배열의 이름은 배열의 시작 주소 값 (첫 번째 요소의 주소값)을 의미
- 다시 말하면, `arr == &arr[0]`

배열과 포인터의 유사점

- 둘 다 (어떤) 주소를 나타냄
- 배열, 포인터 모두 *연산을 사용 가능
- 배열, 포인터 모두 []연산을 사용 가능

```
int arr[] = {5, 10, 15};
```

```
int* parr = arr;
```

```
// 5 5 5 5 를 출력
```

```
printf("%d %d %d %d\n", arr[0], *arr, parr[0], *parr);
```

배열과 포인터의 차이점

- 배열은 포인터가 아니다!
- 배열의 이름에 다른 값을 대입할 수 없다.

```
int arr[3] = {5, 10, 20};  
int num = 30;  
arr = &num; // X
```

컴파일 에러!

error C2106: '=' : 왼쪽 피연산자는 l-value이어야 합니다.

- [배열 이름의 값]과 [배열 이름의 주소값]이 같다.

```
int arr[3] = {5, 10, 20};  
printf("arr: %p\n", arr);  
printf("&arr: %p\n", &arr);
```

```
arr: 1636840  
&arr: 1636840
```

두 값이 같다 (포인터라면 다른 주소값이 나올 것)

배열과 포인터의 차이점

- sizeof 연산자 결과가 다르다.

```
int arr[3] = {5, 10, 20};  
int* parr = arr;  
int size1 = sizeof(arr);  
int size2 = sizeof(parr);
```

size1==12 : 배열 전체의 크기
size2==4 : 포인터 변수의 크기 (32bit)

C Example

```
#include <stdio.h>
int main()
{
    int arr[3] = {5, 10, 20};
    int num = 30;
    //arr = &num;    // compile error
    int* parr = arr;

    printf("%d %d %d %d\n", arr[0], *arr, parr[0], *parr);

    printf("arr: %p\n", arr);
    printf("&arr: %p\n", &arr);

    printf("parr: %p\n", parr);
    printf("&parr: %p\n", &parr);

    printf("size of arr: %d\n", sizeof(arr));
    printf("size of parr: %d\n", sizeof(parr));

    return 0;
}
```

- Python에는 pointer의 개념이 존재하지 않는다.

포인터에 대한 증가/감소 연산

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

```
int main()  
{  
    int i = 1;  
    double d = 1.2;  
    int* pi = &i;  
    double* pd = &d;
```

| |
|---|
| pi: 1636948, pi+1: 1636952, pi+2: 1636956 |
| pd: 1636932, pd+1: 1636940, pd+2: 1636948 |

```
    printf("pi: %p, pi+1: %p, pi+2: %p\n", pi, pi+1, pi+2);  
    printf("pd: %p, pd+1: %p, pd+2: %p\n", pd, pd+1, pd+2);
```

```
    return 0;
```

```
}
```

포인터에 대한 증가/감소 연산

```
int i = 1;  
double d = 1.2;  
int* pi = &i;  
double* pd = &d;
```

```
pi: 1636948, pi+1: 1636952, pi+2: 1636956  
pd: 1636932, pd+1: 1636940, pd+2: 1636948
```

- int형 포인터에 1을 더하면, 실제 값은 4 증가
- double형 포인터에 1을 더하면, 실제 값은 8 증가
- ...
- (특정 자료형) 포인터에 1을 더하면, 실제 값은 sizeof(자료형)만큼 증가
- 감소도 마찬가지로

배열의 [] 연산의 의미

- $\text{arr}[i]$: 배열에서 index i 에 위치하는 요소의 값 ($i+1$ 번째 값)
- 예) `int arr[3] = {5, 10, 20};`
- $\text{arr}[2]$: int형 배열 arr에서 index 2에 위치하는 요소의 값

| | | | | | | | | | | | | | | | |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1638050 | 1638051 | 1638052 | 1638053 | 1638054 | 1638055 | 1638056 | 1638057 | 1638058 | 1638059 | 1638060 | 1638061 | 1638062 | 1638063 | 1638064 | 1638065 |
| | | arr[0] | 5 | | | arr[1] | 10 | | | arr[2] | 20 | | | | |

포인터의 증가/감소 연산의 의미

- $*(arr+i)$: 배열의 시작 주소로부터 i 만큼 증가된 주소에 저장되어 있는 값

- 예) `int arr[3] = {5, 10, 20};`

- $*(arr+2)$: int형 배열 **arr의 시작 주소**로부터 **2 만큼 증가**된 **주소**에 저장되어 있는 값

| | | | | | | | | | | | | | | | |
|---------|---------|-------------|---------|---------|---------|--------------|---------|---------|---------|--------------|---------|---------|---------|---------|---------|
| 1638050 | 1638051 | 1638052 | 1638053 | 1638054 | 1638055 | 1638056 | 1638057 | 1638058 | 1638059 | 1638060 | 1638061 | 1638062 | 1638063 | 1638064 | 1638065 |
| | | arr[0] 5 | | | | arr[1] 10 | | | | arr[2] 20 | | | | | |

포인터의 증가/감소 연산과 배열의 [] 연산의 관계

- 배열에서 index i 에 위치하는 요소의 값 ($i+1$ 번째 값)
- 배열의 시작 주소로부터 i 만큼 증가된 주소에 저장되어 있는 값



$arr[i] == *(arr+i)$

- (arr 이 배열의 이름이어도, 포인터 변수여도 성립)

C Example

```
#include <stdio.h>

int main()
{
    int arr[] = {5, 10, 15, 20};
    int* parr = arr;
    int i;

    for(i=0; i<4; i++)
    {
        // 모두 같은 값을 나타낸다.
        printf("%d %d %d %d\n", arr[i], *(arr+i), parr[i],
*(parr+i));
    }

    return 0;
}
```


배열의 내용을 보기 좋게 출력해주는 함수?

```
int main()
{
    int arr[] = {5, 10, 15};
    printArray(arr);

    return 0;
}
```

```
void printArray(    ?    )
{
    int i;
    for(i=0; i<3; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

배열의 내용을 보기 좋게 출력해주는 함수?

포인터 매개변수로 배열의
시작 주소 값 전달

```
int main()
{
    int arr[] = {5, 10, 15};
    printArray(arr);

    return 0;
}
```

```
void printArray(int* arr)
{
    int i;
    for(i=0; i<3; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

왜 이렇게 하는 것이 가능?

```
int arr[] = {5, 10, 15};
int* parr = arr;
printf("%d %d %d %d\n", arr[1], *(arr+1), parr[1],
*(parr+1));
```

이렇게 배열을 포인터처럼, 포인터를
배열처럼 쓸 수 있으니까...

다양한 길이의 배열을 인자로 받고 싶다면?

- 아래 코드에서 무엇이 문제일까?

```
int main()
{
    int arr[] = {5, 10, 15, 1};
    printArray(arr);

    return 0;
}

void printArray(int* arr)
{
    int i;
    for(i=0; i<3; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

- void printArray(int* arr)로 전달받는 arr은 배열의 시작 주소를 가리키는 포인터이므로 배열의 길이 정보를 가지고 있지 않다.
- 그렇다면 어떻게 해야 할까? -> 배열의 길이도 인자로 전달!

함수의 인자로 배열을 전달하는 방법

- 포인터 매개변수로 배열의 시작 주소 값 전달
- 배열의 길이도 함께 인자로 전달

```
int main()
{
    int arr[] = {5, 10,
15,1};
    printArray(arr, 4);

    return 0;
}
```

```
void printArray(int* arr, int
len)
{
    int i;
    for(i=0; i<len; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

C & Python Examples

- C

```
#include <stdio.h>

void printArray(int* arr, int len)
{
    printf("Array ");
    for(int i=0; i<len; i++)
        printf("[%d]:%d, ", i, arr[i]);
    printf("\n");
}

int main()
{
    int arr[] = {5, 10, 15, 20, 25, 30};
    printArray(arr, sizeof(arr)/sizeof(int));

    return 0;
}
```

- Python

```
def printArray(arr):
    print('Array ', end='')
    for i in range(len(arr)):
        print("[%d]:%d, %(i, arr[i]), end='')
    print()

arr = [5, 10, 15, 20, 25, 30]
printArray(arr)
```

배열을 인자로 전달할 때 또 다른 표기법

```
void printArray(int* arr, int len)
```

- 이렇게 하는 대신

```
void printArray(int arr[], int len)
```

- 이렇게 할 수도 있다.
- 완전히 동일한 의미. arr은 배열이 아닌, 포인터이다.
- 단지 “배열의 느낌”을 좀 더 주는 표기법이라고 할 수 있다.