프로그래밍 기초

11장. 포인터 기초 Part 2

학습목표

- **포인터 변수를 이해하고 설명할 수 있다.**
  - 메모리와 주소, 주소연산자 &
  - *를 사용한 포인터 변수 선언과 간접참조 방법
  - 포인터 변수의 연산과 형변환

- **다중 포인터와 배열 포인터를 이해하고 설명할 수 있다.**
  - 이중 포인터의 필요성과 선언 및 사용 방법
  - 증감연산자와 포인터와의 표현식
  - 포인터 상수

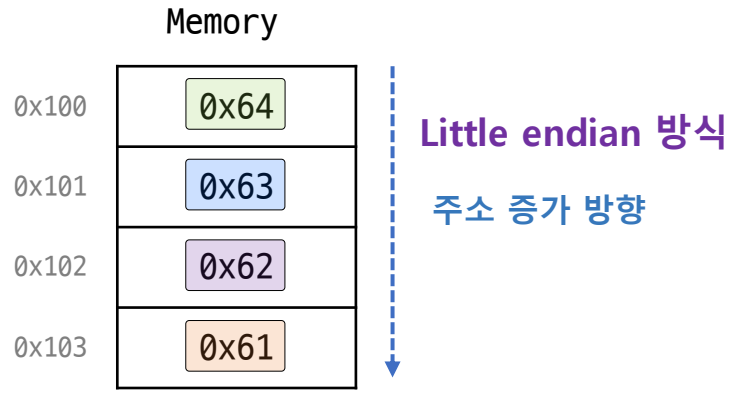
- **배열과 포인터 관계에 대하여 이해하고 설명할 수 있다.**
  - 배열이름은 포인터 상수이며 포인터 변수로도 참조
  - 1차원과 2차원 배열의 배열 포인터 활용
  - 포인터 배열

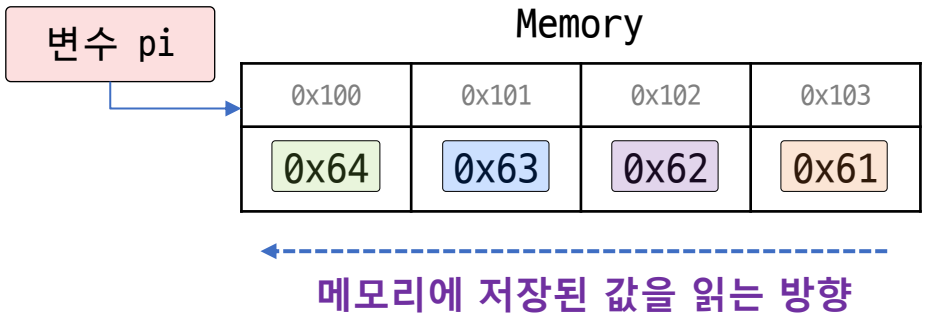
2

변수의 내부 저장 표현 (Little Endian 방식)

- **변수 value에 16진수 0x61626364를 저장**
  - 변수 value의 주소가 100번지
    - ➢ 100번지 1바이트 내부에 16진수 64가 저장
      - – **다음 주소 101번지에는 63이 저장, 다음에 각각 62, 61이 저장**
    - ➢ 가장 작은 값이 가장 작은(낮은) 주소에 저장되는 방식

```
int value = 0x61626364;
int *pi = &value;

printf("%#x %d\n", value, value);
```





변수의 내부 저장 표현 예제

<06little_endian.c>

```c
#include <stdio.h>

int main()
{
    int value = 0x61626364;
    int *pi = &value;
    char *ch = (char *)&value;

    printf("%p %d\n", value, value);
    for (int i = 0; i < 4; i++)
        printf("ch+%d \t\t", i);
    printf("\n----------------------------------------------------------\n");

    for (int i = 0; i < 4; i++)
        printf("%-14p ", ch + i);
    printf("\n----------------------------------------------------------\n");

    for (int i = 0; i < 4; i++)
        printf("0x%x \t\t ", *(ch + i));
    printf("\n");

    return 0;
}
```

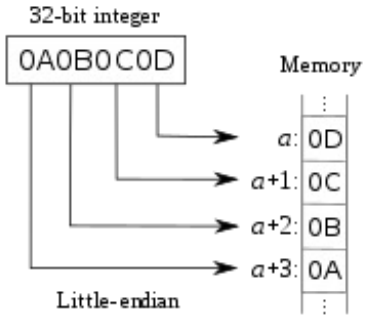
주소 출력

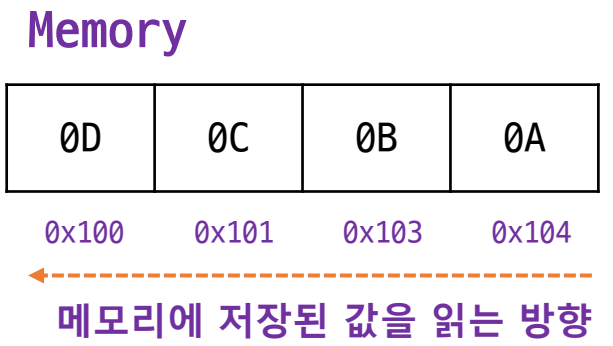
해당 주소의 값 출력

```
0x61626364 1633837924
ch+0            ch+1            ch+2            ch+3
----------------------------------------------------------
0x16fadae88     0x16fadae89     0x16fadae8a     0x16fadae8b
----------------------------------------------------------
0x64            0x63            0x62            0x61
```

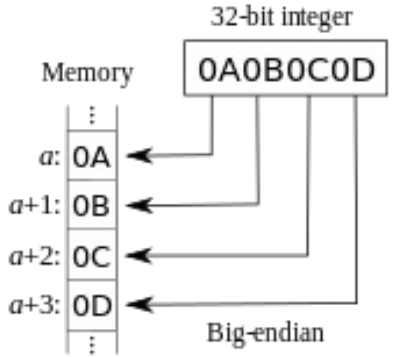
리틀 엔디언, 빅 엔디언

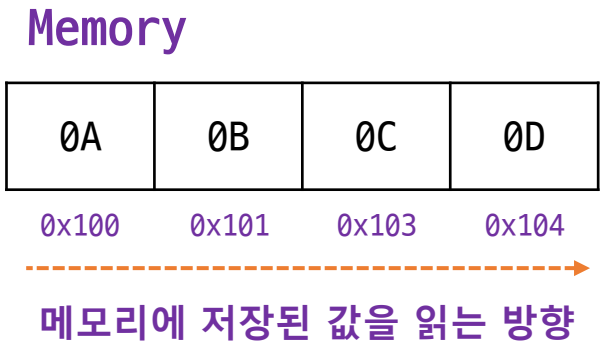
- **리틀 엔디언 (Little Endian)**
  - 낮은 주소(시작 주소)에 하위 바이트부터 기록, Intel CPU, Apple Silicon 계열





- **빅 엔디언(Big Endian)**
  - 낮은 주소(시작 주소)에 상위 바이트부터 기록, Motorola 6800 계열





명시적 형변환

- **포인터 변수는 동일한 자료형끼리만 대입이 가능**
  - 만일 대입문에서 포인터의 자료형이 다르면 경고가 발생
- **포인터 변수는 자동으로 형변환(type cast)이 불가능**
  - 필요하면 명시적으로 형변환을 수행
- ***pc로 수행하는 간접 참조**
  - pc가 가리키는 주소에서부터 1바이트 크기의 char형 자료를 참조
  - *pi는 4바이트인 정수 0x44434241, *pc는 1바이트인 문자코드 0x41을 참조

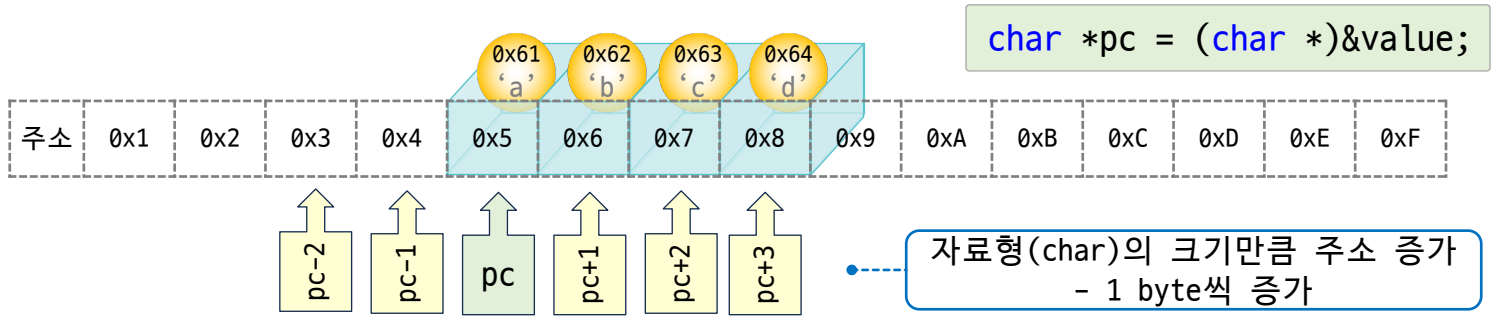
```
int value = 0x44434241; // 정수의 일부분인 코드 41은 문자 'A'
int *pi = &value;
char *pc = (char*)&value;
```

명시적 형변환
(type cast)

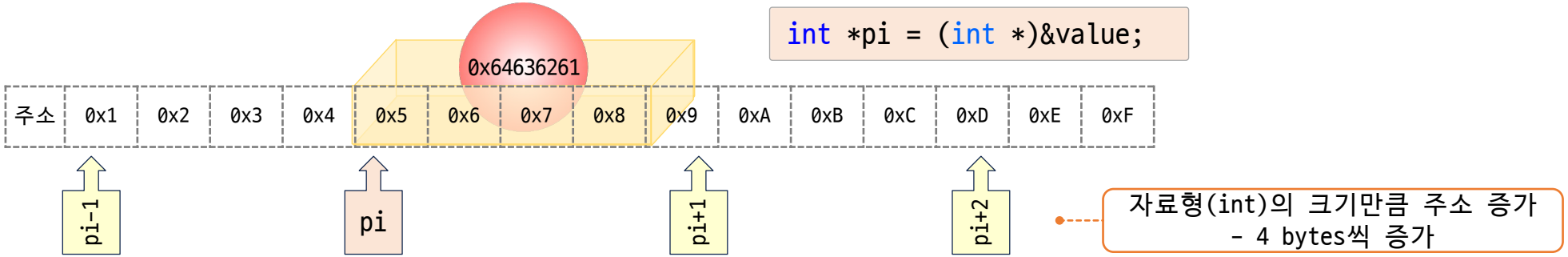
(char*)가 없으면 Warning:
incompatible pointer types 발생

명시적 형변환

- **포인터 변수 증가 및 감소**
  - 지정된 주소로부터 시작해서 변수 자료형의 크기만큼 저장공간을 참조
  - 동일한 메모리의 내용과 주소로부터 참조하는 값이 포인터의 자료형에 따라 달라짐
    - char *pc: 1 바이트 참조, int *pi: 4 바이트 참조




```
int value = 0x64636261;
char *pc = (char*)&value;
int *pi = (int*)&value;
```



포인터 변수 주소 증가 예제

<06ptr_increase.c>

```c
#include <stdio.h>

int main()
{
    int value = 0x64636261; // 0x61: 'a', 0x64: 'd'
    char *pc = (char *)&value;
    int *pi = (int *)&value;

    for (int i = 0; i < 4; i++)
    {
        printf("(pc+%d): %p, *(pc+%d): %c\n", i, (pc+i), i, *(pc + i));
    }
    printf("\n");

    printf("*pi: 0x%x, %d\n", *pi, *pi);
    for (int i = 0; i < 4; i++)
    {
        printf("(pi+%d): %p\n", i, (pi + i));
    }
    return 0;
}
```

자료형(char)의 크기만큼 주소 증가
- 1 byte씩 증가

자료형(int)의 크기만큼 주소 증가
- 4 bytes씩 증가

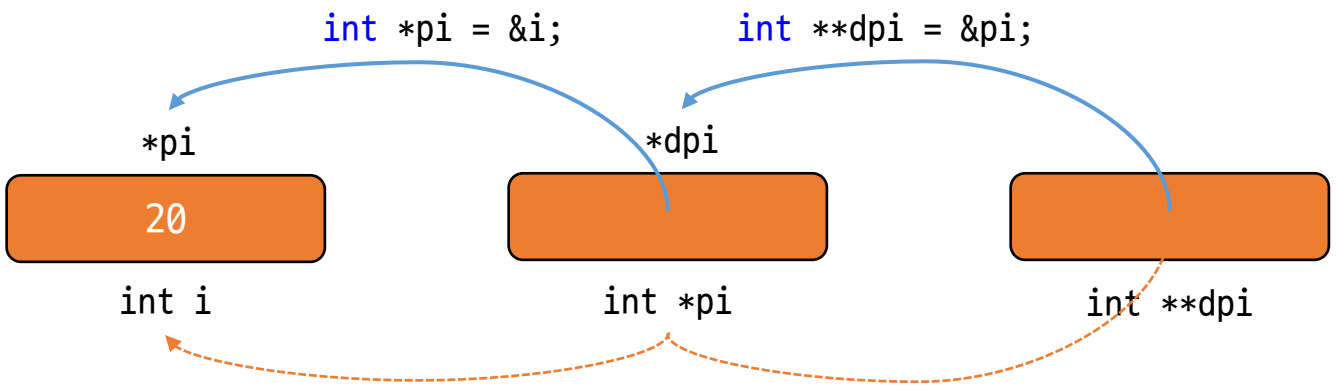
```
(pc+0): 0x16f652a84, *(pc+0): a
(pc+1): 0x16f652a85, *(pc+1): b
(pc+2): 0x16f652a86, *(pc+2): c
(pc+3): 0x16f652a87, *(pc+3): d

*pi: 0x64636261, 1684234849
(pi+0): 0x16f652a84
(pi+1): 0x16f652a88
(pi+2): 0x16f652a8c
(pi+3): 0x16f652a90
```

이중 포인터

- **이중 포인터(double pointer, pointer to pointer)**
  - 포인터 변수의 주소를 가지는 변수: 포인터를 가리키는 포인터:
  - int main(int argc, char **argv)
- **삼중 포인터**
  - 이중 포인터의 주소를 가지는 변수
- **다중 포인터**
  - 변수 선언에서 *를 여러 번 이용하여 다중 포인터 변수를 선언
  - 변수 dpi는 이중 포인터: 포인터 변수 pi의 주소 값을 저장

```
int i = 20;
int *pi = &i;
int **dpi = &pi;
```

```
*pi = i +2;      // i=i+2;
**dpi = *pi + 2; // i=i+2;
```



이중 포인터를 이용한 변수의 참조

- **이중 포인터 변수 dpi**
  - **dpi가 바로 변수 I
- **문장 *pi = i + 30;**
  - 변수 i를 30 증가
- **문장 **dpi = *pi + 30;**
  - 변수 i를 30 증가

<07multptr.c>

```c
#include <stdio.h>

int main(void)
{
    int i = 100;
    int *pi = &i; // 포인터 선언
    int **dpi = &pi;// 이중 포인터 선언

    printf("%p %p %p\n", &i, pi, *dpi);

    *pi = i + 30; // i = i + 30;
    printf("%d %d %d\n", i, *pi, **dpi);

    **dpi = *pi + 30;// i = i + 30;
    printf("%d %d %d\n", i, *pi, **dpi);

    return 0;
}
```

동일한 주소 출력

동일한 값 출력

```
000000000061FE14 000000000061FE14 000000000061FE14
130 130 130
160 160 160
```

간접연산자와 증감 연산자 활용

- **연산자 우선 순위**

우선순위	단항연산자	설명	결합성(계산 방향)
1	a++ a--	후위 증가, 후위 감소	-> (좌에서 우로)
2	++a --a & *	전위 증가, 전위 감소 주소 간접 또는 역참조	<- (우에서 좌로)

- **여러 연산 방법**
  - *p++는 *(p++)으로 (*p)++와 다름
    - 1) p의 주소의 값(*p = 10)
    - 2) p의 주소 증가
  - ++*p와 ++(*p)는 같음
    - p가 가리키는 주소의 값(*p)을 증가
  - *++p와 *(++p)는 같음
    - 1) p의 주소를 증가: (++p)
    - 2) 증가된 주소의 값

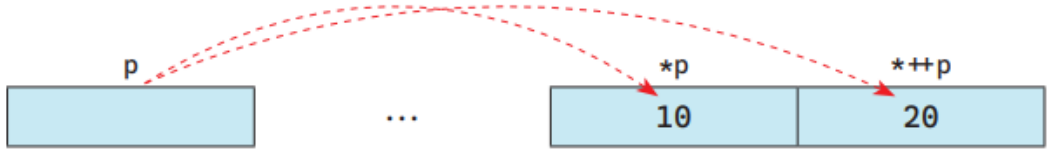


표 11-2 증가연산자 ++와 간접연산자 *의 사용 사례

연산식		결과값		연산 후	
				주소값 p 이동	*p
*p++	*(p++)	10	*p : p의 간접참조 값	p + 1 : p 다음 주소	20
*++p	*(++p)	20	*(p + 1) : (p + 1) 간접참조 값	p + 1 : p 다음 주소	20
(*p)++		10	*p : p의 간접참조 값	p : 변화 없음	11
++*p	++(*p)	11	*p + 1 : *p에 1 더하기	p : 변화 없음	11

포인터 증감 연산 (권장하는 방법 아님)

- 증가 또는 감소 연산

```
int a[] = {10, 20};
int *p = &a[0]; // int *p = a; 동일

int b = *(++p);
```

b = *(p+1);

① p주소를 1 증가: a[1]

② 증가된 p주소의 값: a[1] = 20

b <- 20

```
printf("%d, %d\n", b, *p);
```

b=20, *p=20

```
int a[] = {10, 20};
int *p = &a[0];

int b = *(p++);
```

b = *p;
p++;

① *p의 값을 반환: a[0]

② p주소 1증가: a[1] 가리킴

b <- 10

*p=20 (주소가 1증가한 다음의 값)

```
printf("%d, %d\n", b, *p);
```

b=10, *p=20

연산자	내용	연산자	내용
b = ++a	• a를 1 증가하고 증가된 값을 반환	b = a++	a값을 반환하고, 나중에 1증가
b = --a	• a를 1 감소하고 감소된 값 반환	b = a--	a값을 반환하고, 나중에 1 감소

포인터 증감 연산 예제 (실습)

<06varprtop1.c>

```c
#include <stdio.h>

int main(void)
{
    int a[] = {10, 20};
    int *p = NULL;
    int b = 0;

    p = &a[0];
    b = ++(*p);
    printf("++(*p)= %d, *p= %d, a[0]= %d\n", b, *p, a[0]);
    printf("-------------------------\n");

    p = &a[0];
    b = *(++p);
    printf("1. *(++p)= %d\n", b);

    p = &a[0];
    b = *(p + 1);  // 권장 방법
    printf("1. *(p+1)= %d\n", b);
    printf("-------------------------\n");

    p = &a[0];
    b = *p++;  // b = *(p++); 와 동일
    printf("2. *p++= %d, *p= %d\n", b, *p);

    p = &a[0];
    b = *p;
    p++;  // 권장 방법
    printf("2. (*p; p++)= %d, *p= %d\n", b, *p);

    return 0;
}
```

```
++(*p)= 11
-------------------------
1. *(++p)= 20
1. *(p+1)= 20
-------------------------
2. *p++= 11, *p= 20
2. (*p; p++)= 11, *p= 20
```

포인터 상수: const 사용

- **포인터 상수**
  - 포인터 변수에 const 사용
  - 포인터 변수가 가리키는 변수의 값을 수정할 수 없도록 하는 **상수 선언** 방법

- **포인터 상수 설정 두 가지 방법**

```
int i = 10, j = 20;
```

```
const int* p = &i;
int const* p = &i;
```

p가 가리키는 주소의 값을 변경할 수 없음

```
*p = 20; // 오류 발생
```


```
int* const p = &i;
```

```
p = &j; // 오류 발생
```

p에 저장된 초기 주소를 수정할 수 없음

포인터 상수

- **포인터 변수도 const를 사용해 포인터 상수로 생성**
  - 위치는 세가지(3) 종류가 있지만
  - 첫 번 째와 두 번째는 같은 의미이므로 두 가지 방식이 존재

<09constptr.c>

```c
#include <stdio.h>

int main()
{
    int i = 10, j = 20;
    const int *p = &i; //*p가 상수로 *p로 수정할 수 없음
    //*p = 20; // 오류 발생
    printf("%d ", *p);
    p = &j;
    printf("%d\n", *p);

    double d = 7.8, e = 2.7;
    double *const pd = &d;
    // pd = &e; //pd가 상수로 다른 주소 값을 저장할 수 없음
    printf("%f ", *pd);
    *pd = 4.4;
    printf("%f\n", *pd);

    return 0;
}
```

```
10 20
7.800000 4.400000
```

LAB 표준입력으로 받은 두 실수의 덧셈 (포인터 변수 사용)

- **자료형 double로 선언된 두 변수 x와 y**
  - 표준입력으로 실수를 입력 받아 두 실수의 덧셈 결과를 출력

Lab 11-3 | lab3ptrsum.c | 난이도: ★

```
01  #define _CRT_SECURE_NO_WARNINGS
02  #include <stdio.h>
03
04  int main(void)
05  {
06      double x, y;
07      double* px = &x;
08      double* py = &y;
09
10      // 포인터 변수 px와 py를 사용
11      printf("두 실수 입력: ");
12      scanf("%lf %lf", [          ]);
13      // 합 출력
14      printf("%.2f + %.2f = %.2f\n", [                    ]);
15
16      return 0;
17  }
```

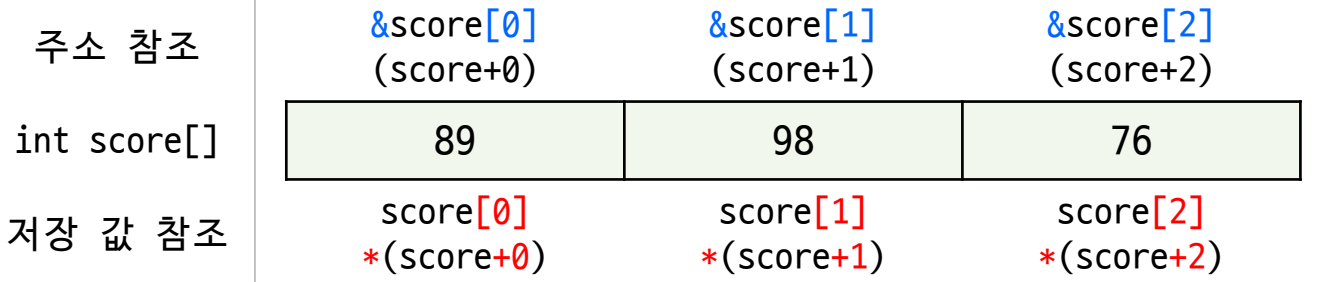
정답

```
12      scanf("%lf %lf", px, py);
14      printf("%.2f + %.2f = %.2f\n", *px, *py, *px + *py);
```

1차원 배열과 포인터

- **배열 이름을 이용한 참조**
  - 배열 이름 자체가 배열의 첫 원소의 주소
    - `score == &score[0]`
  - 배열 이름 score를 이용하여 모든 배열 원소의 주소와 저장 값을 참조 가능
    - 간접 연산자를 이용한 `*score == score[0]`
  - 주소 접근
    - `(score + i) == &score[i]`
  - 간접 연산자를 이용한 값
    - `*(score + i) == score[i]`

```
int score[] = {89, 98, 76};
```



1차원 배열 원소를 접근하는 두 가지 방법 (실습)

- **배열 원소의 주소와 내용 값 다양한 접근 방법**

<10ptrary0.c>

```c
#include <stdio.h>

int main(void)
{
    int a[3] = {5, 10, 15};
    int *p = a; // int *p = &a[0];

    // 간접 연산자 *를 이용한 배열 원소 참조
    printf("%d %d %d\n", *(p+0), *(p+1), *(p+2));

    // 배열의 인덱스를 이용한 배열 원소 참조
    printf("%d %d %d\n", p[0], p[1], p[2]);
    return 0;
}
```

배열 초기화 문장		int score[] = {10, 20, 30};		
원소 값		10	20	30
배열원소 접근 방법	score[i]	score[0]	score[1]	score[2]
	*(score+i)	*score	*(score+1)	*(score+2)
주소값(첫 주소 + 배열원소 크기*i)		100	104 (100 + 1*4)	108 (100 + 2*4)
주소값 접근 방법	&score[i]	&score[0]	&score[1]	&score[2]
	score+i	score	score+1	score+2

<10ptrary1.c>

```c
#include <stdio.h>

void print_array(int *array, int size)
{
    for (int i = 0; i < size; i++)
        printf("%d ", *(array + i));
    printf("\n");

    for (int i = 0; i < size; i++)
        printf("%d ", array[i]);
    printf("\n");
}

int main(void)
{
    int a[3] = {5, 10, 15};

    print_array(a, 3);
    return 0;
}
```

간접 연산자(*) 이용

배열 인덱스[] 이용

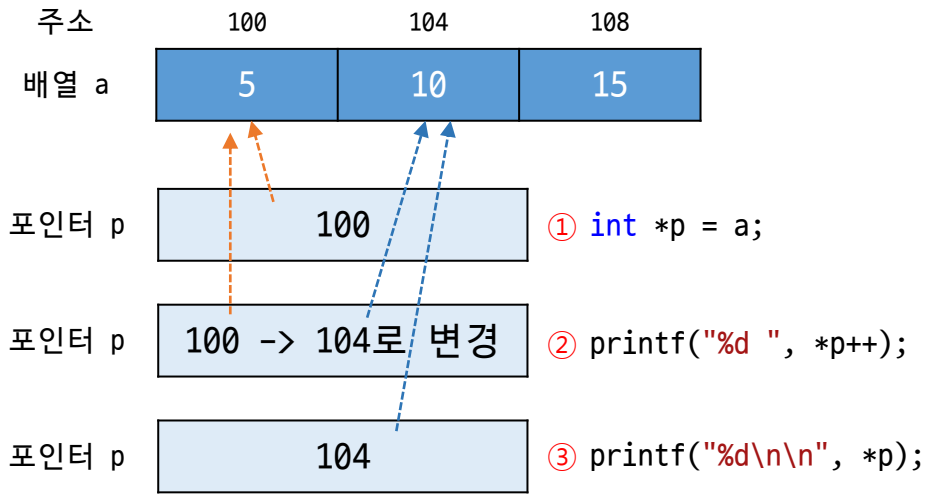
배열의 주소 전달

```
5 10 15
5 10 15
```

포인터 변수와 증감 연산자 활용

- **p++, --p**
  - p가 가리키는 변수의 주소에서 다음이나 이전 저장 공간의 주소로 수정 가능
  - 연산식 *p++는 *(p++)를 의미
    - p가 원래 가리키는 값(*p)인 첫 번째 배열의 저장 값 참조
    - 증가 연산자 p++에 의해 p는 다음 주소값으로 저장
    - 5장의 증감 연산자와 연산자 우선 순위 내용 확인

```
① int a[3] = {5, 10, 15};
① int *p = a; // a == &a[0]

   // a[0]을 출력 후, p 다음 주소로 증가
② printf("%d ", *p++); //*(p++), 5 출력 후, p 다음 주소로 증가

   // a[1]을 출력
③ printf("%d\n\n", *p); // 10 출력
```



1차원 배열에서 배열 이름과 포인터를 사용한 원소와 주소값의 참조

<10ptrary.c>

```c
#include <stdio.h>

int main(void)
{
    int score[] = {10, 20, 30};
    printf("1. %p %p\n", score, (score + 1));
    printf("2. %d %d\n\n", *score, *(score + 1));

    int a[3] = {5, 10, 15};
    int *p = a; // a == &a[0]

    printf("3. %d %d %d\n", *(p), *(p + 1), *(p + 2)); // 배열 원소 값 참조
    printf("4. %d %d %d\n", p[0], p[1], p[2]); // 포인터 변수 p에서 배열처럼 첨자를 사용 가능

    printf("5. %d ", *p++); //a[0] 출력 후, p 주소 증가: printf("%d ", *p); p++; 과 동일
    printf("6. %d\n\n", *p); // a[1]의 값 10 출력

    p = &a[2];
    printf("6. %d ", *p--); // a[2]의 값 출력 후, p 주소 감소: printf("6. %d ", *p); p--;
    printf("%d\n", (*p)--); // a[1]의 값 출력 후, a[1]의 값 1 감소

    printf("7. %d %d %d\n", *(p - 1), *p, *(p + 1)); //p는 a[1]을 가리킴: a[0], a[1], a[2] 출력

    return 0;
}
```

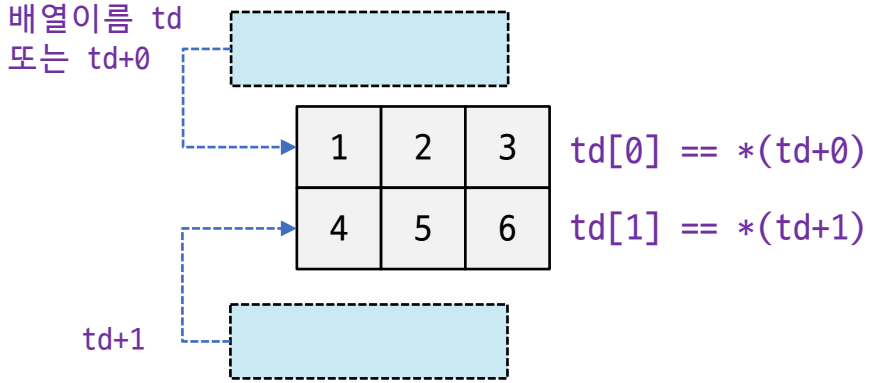
```
1. 0x16bd56e88 0x16bd56e8c
2. 10 20

3. 5 10 15
4. 5 10 15
5. 5 10

6. 15 10
7. 5 9 15
```

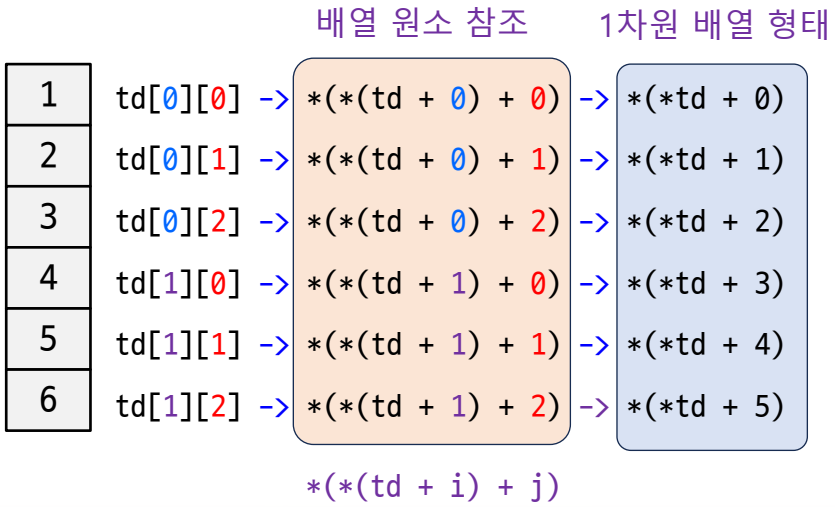
2차원 배열과 간접연산자

- **이차원 배열의 배열이름 td**
  - td는 배열을 대표하며 td[0]를 가리키는 포인터 상수

- **td[0]는 무엇일까?**
  - 배열의 첫 행을 대표
  - 첫 번째 원소 td[0][0]의 주소: &td[0][0]
  - 배열 이름인 td
    - ➢ 포인터의 포인터인 이중 포인터
  - td+1은 두 번째 행을 대표하는 주소 값
  - sizeof(td[0]), sizeof(td[1])
    - ➢ 각각 첫 번째 행과 두 번째 행의 바이트 단위 크기를 반환

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





2차원 배열과 포인터 참조 예제 (실습)

- **연산식 *td+n**
  - 배열의 (n+1)번째 원소의 주소값
- **td[i] + j**
  - (i+1)번째 행과 (j+1)열의 주소
  - (td[i] + j) == &td[i][j]
- **일차원 배열에서 a[i]의 값**
  - *(a+i) == a[i]
- **이차원 배열 td[i][j]의 값**
  - == (td[i]) [j]
  - == *(td[i] + j)
  - == *(*(td + i) + j)

<10ptrary_2.c>

```c
#include <stdio.h>

int main()
{
    int td[][3] = {{1, 2, 3},
                   {4, 5, 6}};

    // 2차원 배열의 시작 주소
    printf("td = %p\n", td);

    // [0]행의 시작 주소
    printf("*(td + 0) = %p, td[0] = %p\n", *(td + 0), td[0]); // &td[0]

    // [1]행의 시작 주소
    printf("td+1          = %p\n", td + 1); // &td[1]과 동일
    printf("*(td + 1) + 0 = %p, td[1] = %p\n", *(td + 1) + 0, td[1]);

    // [0][1]의 주소
    printf("td[0] + 1 = %p, *(td + 0) + 1 = %p, &td[0][1] = %p\n",
           (td[0] + 1), *(td + 0) + 1, &td[0][1]);

    // [1][2]의 주소
    printf("td[1] + 2 = %p, *(td + 1) + 2 = %p, &td[1][2] = %p\n",
           (td[1] + 2), *(td + 1) + 2, &td[1][2]);

    return 0;
}
```

2차원 배열에서 포인터를 사용한 원소와 주소값의 참조: 실습

<11tdary.c>

```c
#include <stdio.h>

#define ROW 2
#define COL 3

int main(void)
{
    int td[][COL] = {{1, 2, 3}, {4, 5, 6}};
    printf("sizeof(td): %zd\n", sizeof(td));
    printf("sizeof(td[0]): %zd, sizeof(td[1]): %zd\n", sizeof(td[0]), sizeof(td[1]));
    printf("sizeof(*td): %zd, sizeof(*(td+1)): %zd\n", sizeof(*td), sizeof(*(td + 1)));
    printf("td : %p, td+1 : %p\n", td, td + 1);
    printf("*td: %p, *(td+1): %p\n", *td, *(td + 1));
    printf("\n");

    for (int i = 0, cnt = 0; i < ROW; i++)
    {
        printf("td[%d]: %p, *(td+%d): %p ", i, td[i], i, *(td + i));
        for (int j = 0; j < COL; j++, cnt++)
            printf("%d %d %d ", *(*td + cnt), *(td[i] + j), *(*(td + i) + j));
        printf("\n");
    }

    **td = 10;              // td[0][0] = 10;
    *(*td + 4) = 20;        // td[1][1] = 20;
    *(*(td + 1) + 2) = 30; // td[1][2] = 30;
    printf("%d %d %d\n", td[0][0], td[1][1], td[1][2]);

    return 0;
}
```

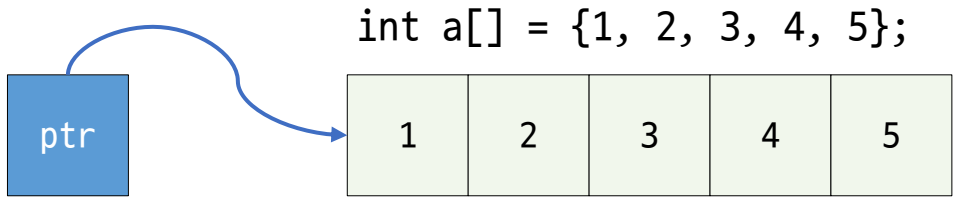
- ****td = 10;**
  - 첫번째 원소 td[0][0]의 값을 10으로 수정
  - td가 이중 포인터
  - *(*td + 0)으로 간접연산자 *이 2개 필요

```
sizeof(td): 24 (6개 원소 x 4bytes)
sizeof(td[0]): 12, sizeof(td[1]): 12
sizeof(*td): 12, sizeof(*(td+1)): 12
td : 0x16d17ae80,   td+1 : 0x16d17ae8c
*td: 0x16d17ae80, *(td+1): 0x16d17ae8c

td[0]: 0x16d17ae80, *(td+0): 0x16d17ae80 1 1 1 2 2 2 3 3 3
td[1]: 0x16d17ae8c, *(td+1): 0x16d17ae8c 4 4 4 5 5 5 6 6 6
10 20 30
```

배열 포인터 (a pointer to an array)

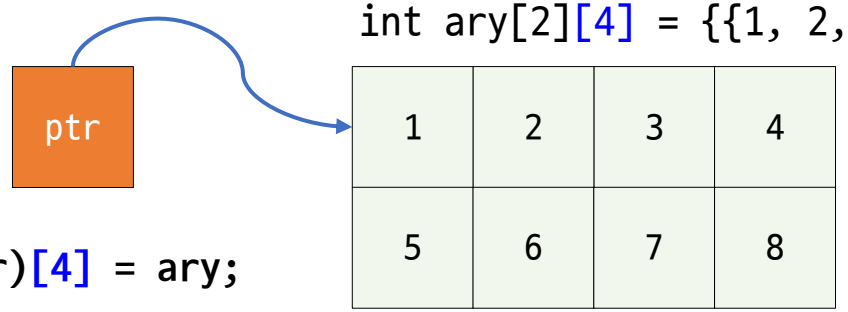
- **배열 포인터: 배열을 가리키는 포인터**
  - 배열의 주소를 가지는 포인터: 배열을 함수 파라미터로 전달 시 사용

1차원 배열 포인터 선언




```
int *ptr = a;
```

2차원 배열 포인터 선언




```
int (*ptr)[4] = ary;
```

`int **ptr = ary`로 사용하지 않음

2차원 배열 포인터

```
int (*ptr)[4];
```

① ()가 우선이므로 ptr은 포인터 선언이 됨

② int[4]를 가리키는 포인터

2차원 배열 포인터 선언

- **2차원 배열 포인터**
  - int (*ptr)[4]; 로 선언
    - 괄호 (*ptr)은 반드시 필요
  - 열(column)이 4인 이차원 배열 ary[][4]의 주소 저장 가능
    - [4]: 이차원 배열에서의 열(column) 크기
    - 이차원 배열의 주소를 저장하는 포인터 변수는 열 크기에 따라 변수 선언이 달라짐

```
원소자료형 *변수이름;
변수이름 = 배열이름;
또는
원소자료형 *변수이름 = 배열이름;
```

```
원소자료형 (*변수이름)[ 배열열크기];
변수이름 = 배열이름;
또는
원소자료형 (*변수이름)[ 배열열크기] = 배열이름;
```

```
int a[] = {8, 2, 8, 1, 3};
int *p = a;
```

```
int ary[][4] = {5, 7, 6, 2, 7, 8, 1, 3};
int (*p)[4] = ary;   //열이 4인 배열을 가리키는 포인터
//int *p[4] = ary;   //포인터 배열
```

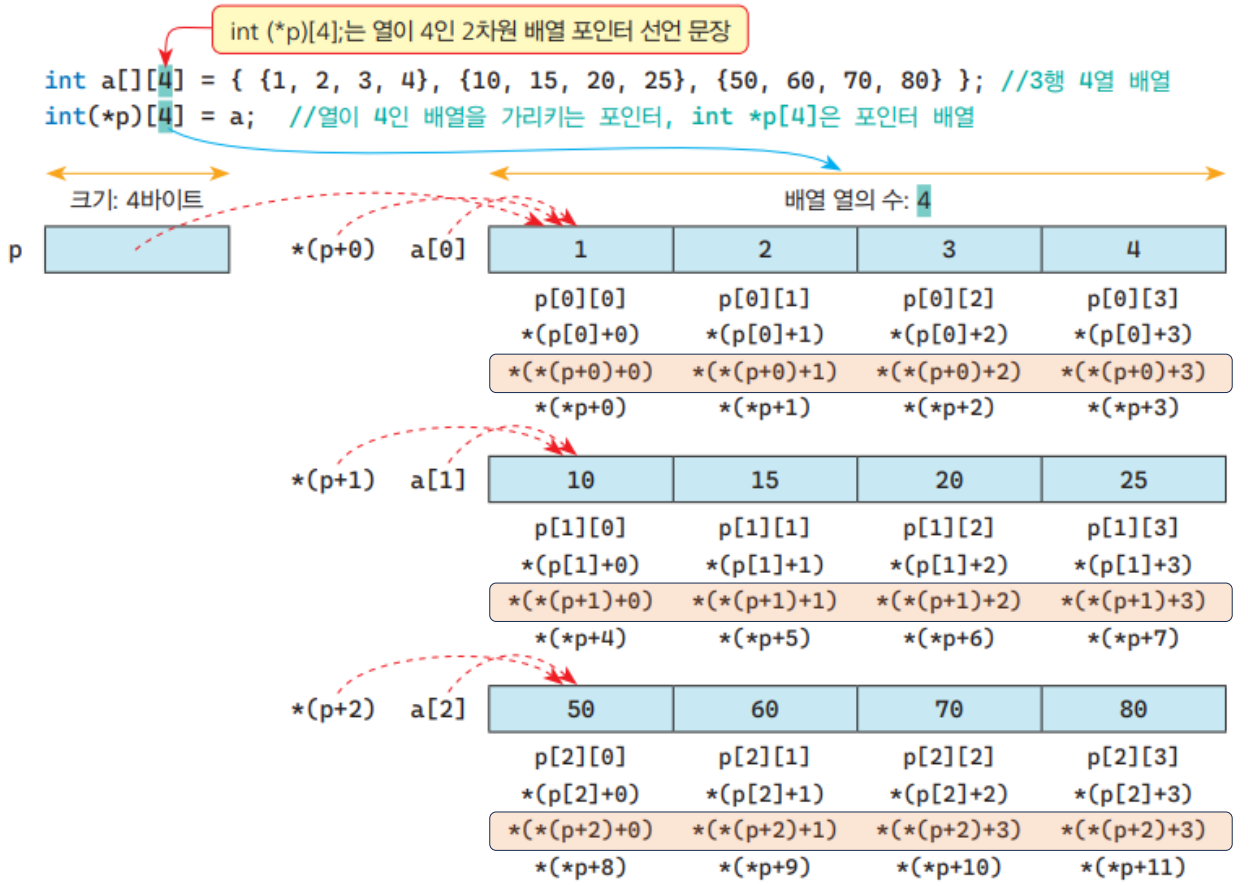
  - 주의: int *ptr[4]; (포인터 배열)
    - int형 포인터 변수를 4개 선언하는 포인터 배열 선언 ➡

```
int *ptr[4];
```

2차원 배열 포인터 선언

- **열이 4인 이차원 배열 포인터 p는 배열 첫 번째 원소의 주소 값**
  - 배열 첫 원소를 참조하려면 **p를 이용

- **괄호가 없는 int *p[4];은 다음에 배울 int형 포인터 변수 4개를 선언하는 포인터 배열 선언 문장**이다.



2차원 배열을 가리키는 배열 포인터의 선언과 이용

- **이차원 배열 배열이름 ary와 배열 포인터 ptr**
  - 연산자 sizeoof 결과 값은 서로 다름
    - sizeof(a): 배열의 총 크기 32= 2rows * 4columns * 4bytes
    - sizeof(ptr): 단순히 포인터의 크기인 8(64비트 시스템인 경우)

- **연산식 *(ptr[i] + j)**
  - (i+1) 행, (j+1)열 원소 참조

- **연산식 *(*(ptr + i) + j)**
  - (i+1) 행, (j+1)열 원소 참조: ptr[i][j] 사용 가능

- **연산식 **ptr++**
  - 연산 우선순위에 따라 **(ptr++)
  - 현재 포인터가 가리키는 원소를 참조하고
    - p을 하나 증가시켜
    - 다음 행의 첫 원소를 가리키게 하는 연산식

Prj12 12tdaryptr.c 2차원 배열을 가리키는 배열 포인터의 선언과 이용 난이도: ★★

```c
#include <stdio.h>

int main(void)
{
   int ary[][4] = { {10, 20, 30, 40}, {50, 60, 70, 80} }; //2행 4열 배열
   int(*ptr)[4] = ary;    //열이 4인 배열을 가리키는 포인터, int *ptr[4]은 포인터 배열

   printf("%zd %zd\n", sizeof(ary), sizeof(ptr));
   printf("%zd %zd\n\n", sizeof(ary[0]), sizeof(ptr[0]));

   printf("%2d %2d\n", **ary, **ptr); //첫 번째 원소, 10
   printf("%2d %2d\n", **(ary + 1), *ary[1]); //두 번째 해의 첫 원소, 50
   printf("%2d %2d\n", **(ptr + 1), *ptr[1]); //두 번째 해의 첫 원소, 50
   printf("%2d %2d\n", *(ary[1] + 1), *(ptr[1] + 1)); //2행 2열, 60
   printf("%2d %2d\n\n", *(*(ary + 1) + 3), *(*(ptr + 1) + 3)); //2행 4열, 80

   printf("%2d ", **ptr++); //배열의 첫 원소 10 참조 후, ptr의 다음 행으로 주소 수정
   printf("%2d\n", **ptr); //두 번째 행의 첫 원소 50 참조

   return 0;
}
```

포인터의 크기는 8바이트

```
32 8
16 16

10 10
50 50
50 50
60 60
80 80

10 50
```

2차원 배열을 가리키는 배열 포인터의 선언과 이용 예제

<12tdaryptr.c>

```c
#include <stdio.h>

int main(void)
{
    int ary[][4] = {{10, 20, 30, 40},
                    {50, 60, 70, 80}};
    int (*ptr)[4] = ary; // column의 크기가 4인 배열을 가리키는 포인터

    printf("sizeof(ary): %zd, sizeof(ptr): %zd\n", sizeof(ary), sizeof(ptr));
    printf("sizeof(ary[0]: %zd, sizeof(ptr[0]): %zd\n\n",
                    sizeof(ary[0]), sizeof(ptr[0]));

    printf("%2d, %2d\n", **ary, **ptr); // 첫 번째 원소, 10
    printf("%2d, %2d\n", **(ary + 1), *ary[1]); // 두 번째 행의 첫 원소
    printf("%2d, %2d\n", **(ptr + 1), *ptr[1]); // 두 번째 행의 첫 원소
    printf("%2d, %2d\n", *(ary[1] + 1), *(ptr[1] + 1)); // 2행 2열, 60
    printf("%2d, %2d\n\n", *(*(ary + 1) + 3), *(*(ptr + 1) + 3));

    printf("%2d ", **ptr++); // 배열의 첫 원소 10 참조 후, ptr의 다음 행으로 주소 수정
    printf("%2d\n", **ptr); // 두 번째 행의 첫 원소 50 참조

    return 0;
}
```

```
sizeof(ary): 32, sizeof(ptr): 8
sizeof(ary[0]: 16, sizeof(ptr[0]): 16

10, 10
50, 50
50, 50
60, 60
80, 80

10, 50
```

가장 일반적인 2차원 배열 포인터 표현 방법

```c
printf("**ptr: %d, ", **ptr);
ptr++; 로 변경 가능
```

2차원 배열 전달 함수 예제 (실습)

<2darray_func.c>

```c
#include <stdio.h>

#define ROWS 5
#define COLS 5
void random_array1(int (*matrix)[COLS], int row, int col)
{
    int num = 1;
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            matrix[i][j] = num++; // 배열 형태로 접근
            printf("%3d ", matrix[i][j]);
        }
        printf("\n");
    }
}

void random_array2(int (*matrix)[COLS], int row, int col)
{
    for (int i = 0; i < row; i++){
        for (int j = 0; j < col; j++){
            *(*(matrix + i) + j) = matrix[i][j] + 1; // 포인터로 접근
            printf("%3d ", *(*(matrix + i) + j));
        }
        printf("\n");
    }
}
```

int *matrix[COLS]와 다름

2차원 배열 포인터 선언
(*배열이름)[column] 형태로 선언

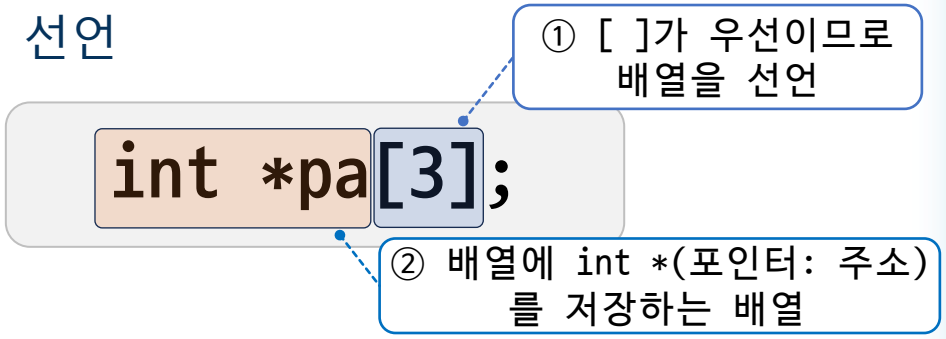
```c
int main()
{
    int array1[ROWS][COLS] = {0};

    printf("Array Access\n");
    random_array1(array1, ROWS, COLS);

    printf("\nIndirect(Pointer) Access\n");
    random_array2(array1, ROWS, COLS);

    return 0;
}
```

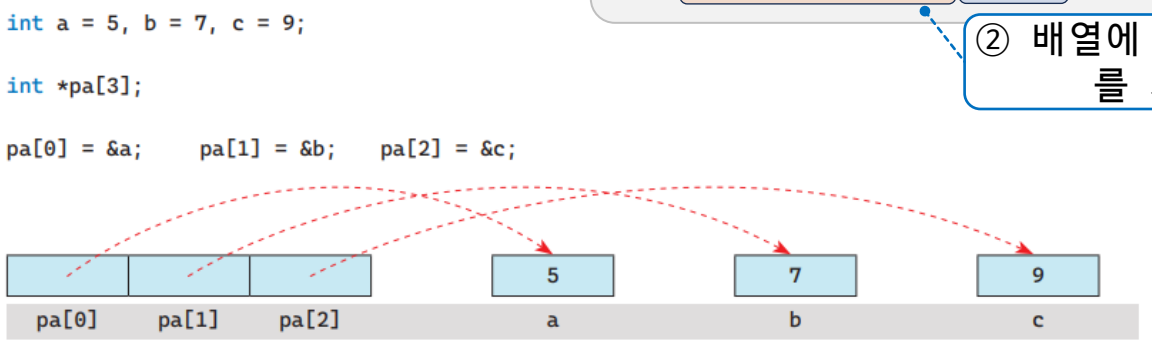
2차원 배열을 전달
- 2차원 배열의 이름만 전달

```
Array Access
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15
 16  17  18  19  20
 21  22  23  24  25

Indirect(Pointer) Access
  2   3   4   5   6
  7   8   9  10  11
 12  13  14  15  16
 17  18  19  20  21
 22  23  24  25  26
```

포인터 배열 개요와 선언

- **포인터 배열(array of pointer)**
  - 포인터(주소)를 저장하는 배열
  - 일반 배열 선언에서 변수 이름 앞에 *를 붙이면 포인터 배열 선언
- **int *pa[3]**
  - 배열 크기가 3인 포인터 배열
  - pa[0] = &a
    - 변수 a의 주소를 저장
  - pa[1]: 변수 b의 주소를 저장
  - pa[2]: 변수 c의 주소를 저장





- **double *dary[5] = {NULL};**
  - NULL 주소를 하나 지정, 나머지 모든 배열원소에 NULL 주소가 지정
  - 문장 float *ptr[3] = {&a, &b, &c};
    - 변수 주소를 하나 하나 직접 지정하여 저장 가능

포인터 배열 변수 선언

자료형 *변수이름[배열크기] ;

```
int *pary[5];
char *ptr[4];
float a, b, c;
double *dary[5] = {NULL};
float *ptr[3] = {&a, &b, &c};
```

여러 포인터를 저장하는 포인터 배열의 선언과 이용

<13aryptr.c>

```c
#include <stdio.h>

#define SIZE 3

int main(void)
{
    // 포인터 배열 변수선언
    int *pary[SIZE] = {NULL};
    int a = 10, b = 20, c = 30;

    pary[0] = &a;
    pary[1] = &b;
    pary[2] = &c;
    for (int i = 0; i < SIZE; i++)
        printf("*pary[%d] = %d\n", i, *pary[i]);

    for (int i = 0; i < SIZE; i++)
    {
        scanf("%d", pary[i]);
        printf("%d, %d, %d\n", a, b, c);
    }
    return 0;
}
```

포인터 배열: 변수의 주소 저장

*pary[i]
- 연결된 변수의 값

pary[i]는 변수의 주소를 저장하고 있으므로 & 가 필요 없음

scanf()를 호출해서 paray[]가 참조하는 변수의 값을 변경

- **포인터 배열 pary**
  - *pary[i]: 변수 a, b, c의 값
  - pary[i]: scanf()로 저장할 주소

```
*pary[0] = 10
*pary[1] = 20
*pary[2] = 30
1
1, 20, 30
2
1, 2, 30
3
1, 2, 3
```

정수 입력하고 Enter 키 입력

LAB 2차원 배열과 배열 포인터 활용

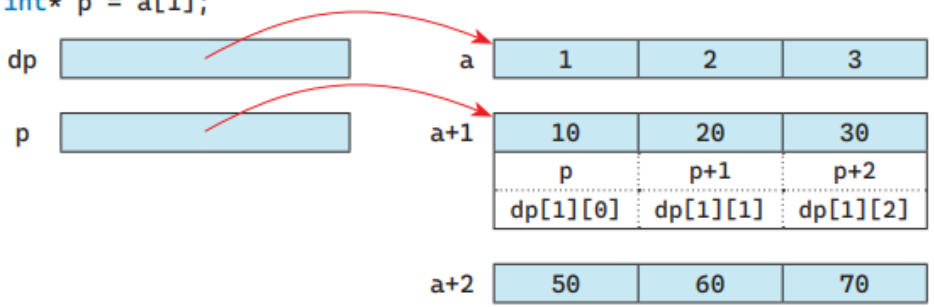
- a: 열이 3인 이차원 배열
- dp: 이차원 배열 포인터
- p: 정수 포인터
  - 이차원 배열 2행을 가리키는 포인터
  - p, p+1, p+2가 2행의 3개 원소, 각각의 주소
  - 이차원 배열 dp로는 dp[1][0], dp[1][1], dp[1][2]가 2행의 3개 원소 값

lab4ptrs.c 난이도: ★★

```
01  #include <stdio.h>
02
03  int main(void)
04  {
05      int a[][3] = { {1, 2, 3}, {10, 20, 30}, {50, 60, 70} };
06
07      int(*dp)[3] = a;
08      int* p =
09
10      printf("%d %d %d\n",                    );
11      printf("%d %d %d\n", dp[1][0], dp[1][1], dp[1][2]);
12
13      return 0;
14  }
```

```
08      int* p = a[1];
10      printf("%d %d %d\n", *p, *(p+1), *(p+2));
```

```
int a[][3] = { {1, 2, 3}, {10, 20, 30}, {50, 60, 70} };

int (*dp)[3] = a;
int* p = a[1];
```



Questions?