
C 프로그래밍 및 실습

9. 포인터

세종대학교

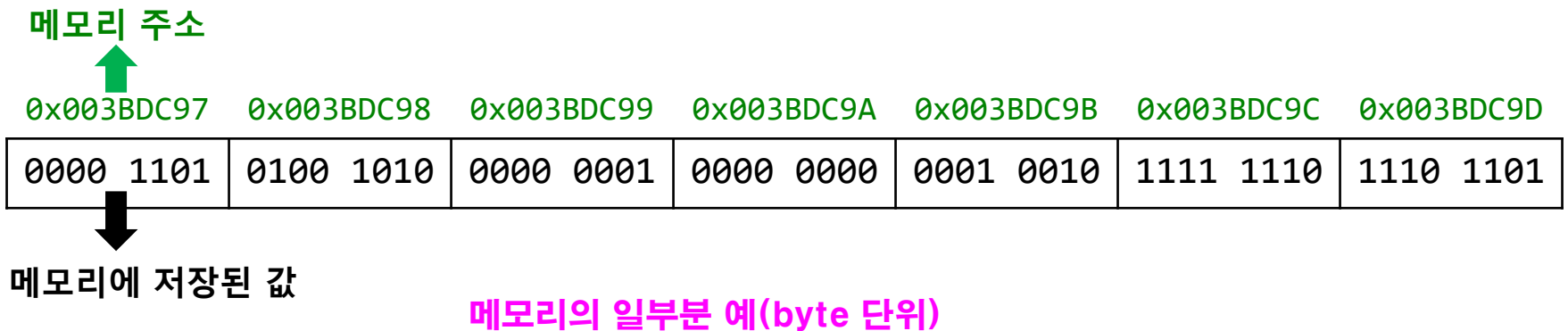
목차

- 1) 포인터 개요
- 2) 포인터 선언과 사용
- 3) 배열과 포인터
- 4) 포인터 연산
- 5) 포인터 인자와 주소 반환
- 6) 포인터 배열
- 7) 다중 포인터

1) 포인터 개요

■ 메모리

- 프로그램이 실행되기 위해 필요한 정보(값)을 저장하는 공간
- 1 byte (8 bits) 단위로 물리 주소가 부여 되어 있음
- 개념적으로, 메모리는 일렬로 연속되어 있는 크기가 1 byte 인 방들의 모음이라고 볼 수 있음
- 일반적으로 주소의 길이는 4 bytes이고, 주소는 16진수로 표현



1) 포인터 개요

▪ 변수와 메모리의 관계

- 변수는 선언될 때, 메모리에 그 변수를 위한 공간이 할당됨
✓ (주의) 변수에 할당되는 메모리의 주소는 시스템마다 다르다.
- **주소연산자(&)** : 변수에 할당된 메모리 공간의 시작 주소를 구해줌

```
int a = 0;  
printf("%d, %p", a, &a); // %p: 주소를 16진수로 출력
```

결과:

0, 003BDC98

⇒ 주소 값은 다르게 나올 수 있음

주소

0x003BDC97	0x003BDC98	0x003BDC99	0x003BDC9A	0x003BDC9B	0x003BDC9C	0x003BDC9D
0000 1101	0000 0000	0000 0000	0000 0000	0000 0000	1111 1110	1110 1101

a에 할당된 메모리 공간 (4bytes) : 한번 할당되면 고정됨

1) 포인터 개요

- C프로그램에서 변수의 의미는 (2가지)

1. 그 변수에 **할당된 공간**을 의미 (주소를 뜻하는 것은 아님)

- ✓ 선언 or 대입문의 왼쪽 변수(l-value)로 사용될 때

2. 그 변수에 **저장된 값**을 의미

- ✓ 대입문의 오른쪽 변수(r-value), 조건식, 함수의 인수로 사용될 때

```
char c1, c2;    // c1, c2를 위한 공간을 메모리에 할당
c1 = c2;        // c1에 c2를 저장 → c1의 공간에 c2에 저장된 값을 저장
if( c1 < c2 )   // c1이 c2보다 작으면
                // → c1에 저장된 값이 c2에 저장된 값보다 작으면
printf("%c",c1); // c1을 인수로 전달 → c1에 저장된 값을 전달
```

주소

0x003BDC97 0x003BDC98 0x003BDC99 0x003BDC9A 0x003BDC9B

0000 1101	0000 0000	0000 0000	0000 0000	0001 1010
-----------	-----------	-----------	-----------	-----------

c1

c2

1) 포인터 개요

- [예제 9.1] 아래와 같이 선언된 변수들의 주소를 출력하고, 출력된 주소를 보고 메모리에 변수가 할당된 모습을 그림으로 그려보자.

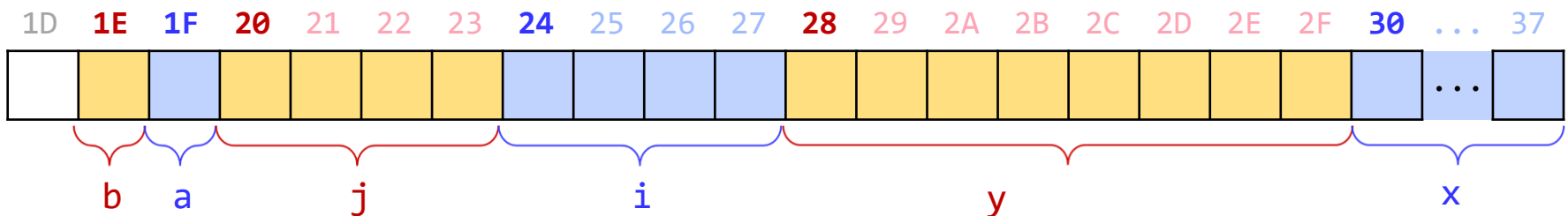
출력 예시

```
char a, b;  
int i, j;  
double x, y;
```

```
a: 0018F91F, b: 0018F91E  
i: 0018F924, j: 0018F920  
x: 0018F930, y: 0018F928
```

- 메모리에 변수가 할당된 모습
✓ 연속으로 할당된다는 보장은 없음

주소 (마지막 두 자리만 표시)



1) 포인터 개요

- [예제 9.2] 아래와 같이 배열 원소들의 주소를 출력하고, 출력된 주소를 보고 메모리에 변수가 할당된 모습을 그림으로 그려보자.

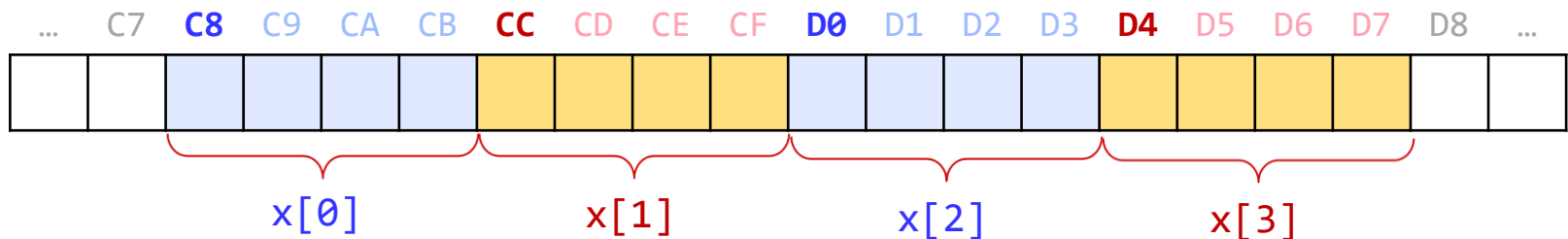
출력 예시

```
int x[4];
```

```
x[0]: 001FFEC8  
x[1]: 001FFECB  
x[2]: 001FFED0  
x[3]: 001FFED4
```

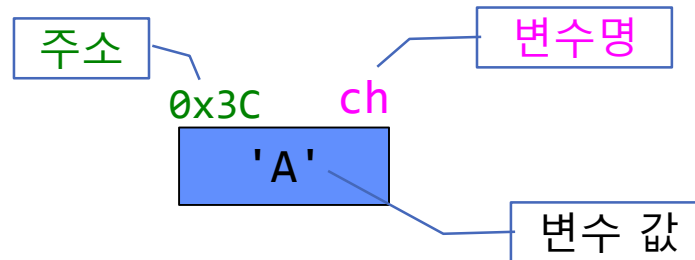
- 메모리에 배열이 할당된 모습
✓ 배열은 항상 연속된 공간에 할당됨

주소 (마지막 두 자리만 표시)



1) 포인터 개요

- 포인터 (자료형)
 - 주소를 나타내는 특수 자료형
 - 주소는 기본적으로 양의 정수로 표현됨
하지만, int(정수형 자료형)와 구별되어 처리됨 (다른 자료형)
 - 변수를 나타내는 메모리 그림



목차

- 1) 포인터 개요
- 2) 포인터 선언과 사용**
- 3) 배열과 포인터
- 4) 포인터 연산
- 5) 포인터 인자와 주소 반환
- 6) 포인터 배열
- 7) 다중 포인터

2) 포인터 선언과 사용

- 포인터 (변수) 선언

- 구문: 변수 명 앞에 * (참조연산자)만 덧붙이면 됨

✓ 기존의 자료형 표시 + 포인터라는 표시

✓ 예)

```
char    *pch;  
int     *pnum;
```

- ✓ pch와 pnum은 똑같이 주소를 저장하지만
대상의 자료형이 다르기 때문에 **다른 자료형**으로 취급

- ✓ pch는 **문자형 포인터** (변수)이고 pnum은 **정수형 포인터** (변수)

2) 포인터 선언과 사용

- 초기화

- 일반 변수 초기화 형태와 동일

```
int num, *pnum = &num;
```

- ✓ (주의!!) num이 먼저 선언 되어야 함

```
int *pnum = &num, num;           // 컴파일 오류
```

- 포인터 변수 선언의 다양한 형태

- ✓ 동일 기본 자료형(int)에서 파생된 자료형의 변수는 모아서 선언 가능

```
int *pnum1, num1=10, *pnum2, num2, arr[10];
```

- ✓ 그러나, 가독성 때문에 추천 안 함

2) 포인터 선언과 사용

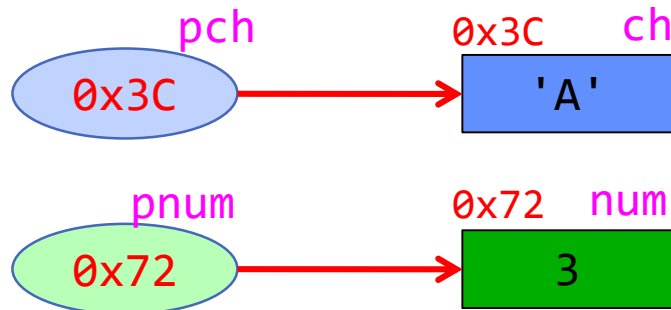
• 포인터 대입 (연결)

- 포인터 (변수)에 **주소를 대입**하여 특정 변수와 연결시키는 것을 **"가리킨다"**라고 표현하고, 그림에서는 **화살표 →**로 표시

```
char ch = 'A', *pch;  
int num = 3, *pnum;
```

pch = &ch; ⇨ pch에 변수 ch의 주소 대입(연결)
pnum = # ⇨ pnum에 변수 num의 주소 대입(연결)

```
printf("%p %c\n", pch, ch); ⇨ %p: 주소 출력 서식  
printf("%p %d\n", pnum, num);
```



실행 예시

```
001EA03C A  
001EA072 3
```

2) 포인터 선언과 사용

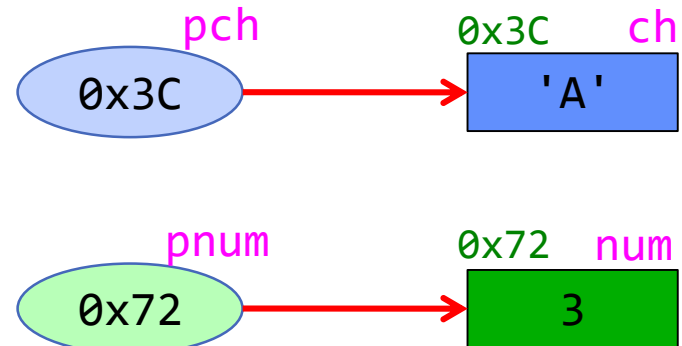
■ 포인터 참조

- 포인터 (변수)가 가리키는 변수에 접근하는 것
- **참조 연산자 *** (간접연산자, 포인터연산자라고도 부름)를 사용
예) `*pch` : 포인터 `pch`가 가리키는 변수, `0x3C`번지에 저장된 값

```
char ch = 'A';  
char *pch = &ch;  
  
int num = 3, *pnum = &num;  
  
printf("%p %c\n", pch, *pch);  
printf("%p %d\n", pnum, *pnum);
```

실행 예시

```
001EA03C A  
001EA072 3
```



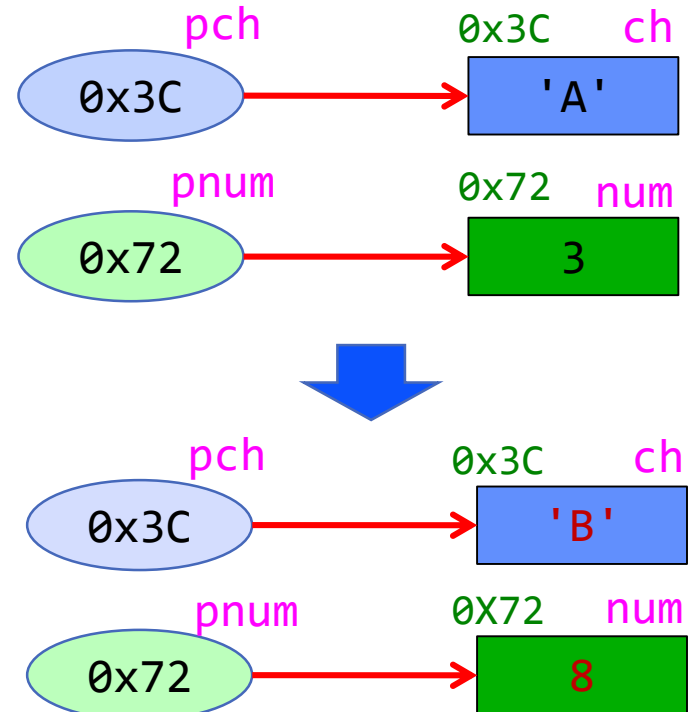
2) 포인터 선언과 사용

- 참조연산자를 이용한 대입 예시
 - ✓ `*pch = 'B'` 의 의미: `pch`가 가리키는 공간에 'B' 대입
 - ✓ `*pch = 'B'`는 `ch = 'B'`와 동일한 기능 수행
전자는 **간접 접근**, 후자는 **직접 접근**

```
char ch = 'A', *pch = &ch;  
int num = 3, *pnum = &num;  
  
*pch = 'B';  
*pnum += 5;  
  
printf("%c %d\n", ch, num);
```

실행 결과

B 8



2) 포인터 선언과 사용

- 참조연산자 추가 예시
 - ✓ *pnum은 정수를 나타내므로, 정수를 사용하는 어떤 형태든 가능
 - ✓ 단, 참조연산자와 다른 연산자와의 우선 순위에 주의해서 사용

```
int num = 3, *pnum = &num;
```

```
*pnum = *pnum / 2 + 4; ⇨ 정수 연산: num에 num/2+4 = 5 대입
```

```
if( *pnum == 5)          ⇨ 정수 비교: ( num == 5 )
```

```
    ++*pnum;             ⇨ 정수 연산: ++(*pnum) ⇨ ++num
```

```
printf("%d", *pnum);     ⇨ 함수의 인자로 사용
```

실행 결과

6

2) 포인터 선언과 사용

- [예제 9.3] 1단계: 다음에 해당하는 문장들을 차례로 작성하고,
2단계: 메모리 그림을 그려보시오. (그림1 ~ 그림6)
 - ✓ (그림 1) int 변수 num1, num2 선언, int 포인터 변수 p 선언 및 num1의 주소로 초기화 (하나의 문장으로)
 - ✓ (그림 2) p가 가리키는 변수에 3000 대입
 - ✓ (그림 3) num2에 p가 가리키는 변수값 대입
 - ✓ (그림 4) p가 num2를 가리키도록 변경
 - ✓ (그림 5) p에 연결된 변수에 p가 가리키는 변수 값 - 1000 대입
 - ✓ (그림 6) num1에 p가 가리키는 변수 값의 2배 대입
 - ✓ num1, num2, p를 출력한다.
 - ✓ num1의 주소, num2의 주소, p의 주소를 출력한다.

2) 포인터 선언과 사용

```
int num1, num2, *p = &num1;
```

```
*p = 3000;
```

```
num2 = *p;
```

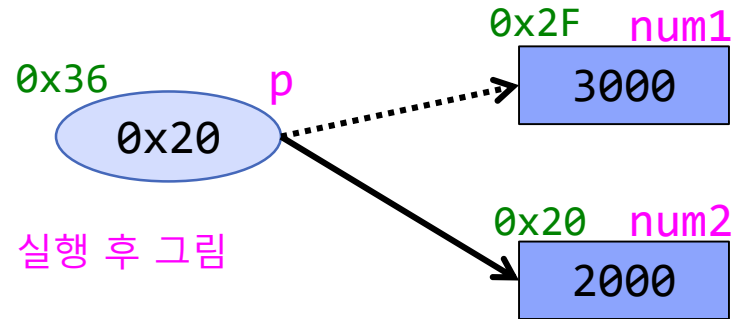
```
p = &num2;
```

```
*p = *p - 1000;
```

```
num1 = *p * 2; // 첫 번째 *은 참조연산자, 두 번째 *은 곱셈연산자
```

```
printf("값: num1=%d num2=%d p=%p\n", num1, num2, p);
```

```
printf("주소: num1=%p num2=%p p=%p\n", &num1, &num2, &p);
```

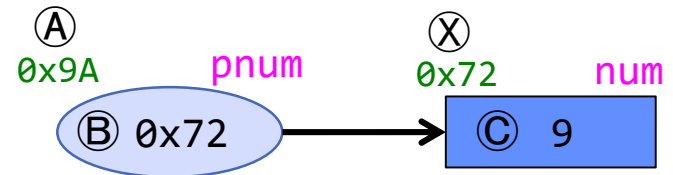


2) 포인터 선언과 사용

- 포인터와 관련한 두 연산자 정리 (pnum 기준)
 - 주소연산자(&): 해당 변수의 주소 값 (그림의 ㉠)
 - 변수 이름: 변수 영역 또는 변수에 저장된 값 (그림의 ㉡)
 - 참조연산자(*): 포인터가 가리키는 변수(그림의 ㉢)
- (주의) pnum과 &num의 값은 동일하지만, **지칭하는 부분은 전혀 다름**
- 질문) &*pnum 과 *&num이 각각 의미하는 부분은?

```
int num = 9, *pnum = &num;
```

```
printf("%p\n", &pnum);  ⇒ ㉠ 0x9A
printf("%p\n", pnum);   ⇒ ㉡ 0x72
printf("%d\n", *pnum);  ⇒ ㉢ 9
printf("%p\n", &num);   ⇒ ㉣ 0x72
```

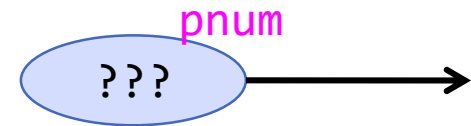


포인터를 이해하고 학습하기 위한 가장 좋은 방법은
메모리 그림을 그리는 것이다!!

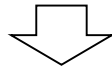
2) 포인터 선언과 사용

- 포인터 주의사항 1 (초기화)

- 선언 후 연결 없이 바로 사용하면?



```
int *pnum ;    // pnum에는 쓰레기 값  
*pnum = 9 ;    // 런타임(실행) 오류 발생
```



```
int *pnum, num;  
pnum = &num; // 반드시 어떤 변수에 연결 후 사용  
*pnum = 9;
```

- 널(NULL) 포인터

- ✓ 주소 값 0을 나타내는 특별한 기호로 아무것도 가리키지 않음을 의미
- ✓ NULL의 값은 0이므로, 조건문에서 사용하면 거짓에 해당
- ✓ 예기치 못한 오류 방지를 위해 포인터 변수를 NULL로 초기화

```
int *pnum = NULL;
```

2) 포인터 선언과 사용

▪ 포인터 주의사항 2

- & (주소연산자)는 포인터를 포함한 모든 변수에 사용가능
- * (참조연산자)는 포인터 변수에서만 가능
 - ✓ *num (num이 가리키는 변수) 은 정의 되지 않음

```
int num=9, *pnum = &num;

printf("%p %p %d\n", &pnum, pnum, *pnum);

printf("%p %d %d\n", &num, num, *num); // 컴파일 오류
```

▪ 포인터 주의사항 3 (대입)

- 포인터의 자료형과 연결된 변수의 자료형은 일치해야 한다.
- 서로 다른 자료형의 포인터 간 대입
 - ✓ 문법적으로는 허용이 되기도 하지만 (컴파일 경고만 발생)
 - ✓ 프로그램 오류의 원인이 됨

2) 포인터 선언과 사용

- 포인터의 크기

- 포인터의 종류(자료형)에 관계없이 주소를 저장하기 위해 필요한 공간은 동일
 - ✓ 단, 포인터의 크기는 시스템에 따라 다를 수는 있음
- sizeof 연산자를 이용하여 확인해보자.

```
char *pch;  
int *pnum;  
double *pdnum;  
  
printf("%d\n", sizeof(pch));  
printf("%d\n", sizeof(pnum));  
printf("%d\n", sizeof(pdnum));
```

실행 결과

4
4
4

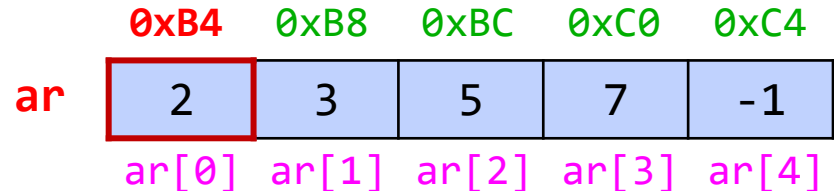
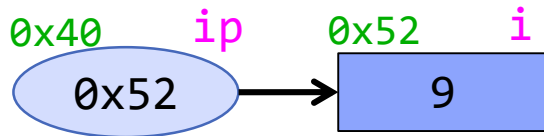
목차

- 1) 포인터 개요
- 2) 포인터 선언과 사용
- 3) **배열과 포인터**
- 4) 포인터 연산
- 5) 포인터 인자와 주소 반환
- 6) 포인터 배열
- 7) 다중 포인터

3) 배열과 포인터

- 배열 이름의 비밀 (예: 대입문 오른쪽)
 - 배열 이름은 배열의 0번 원소의 시작 주소를 의미한다. (특별하다)
 - ✓ 비교) &ar는 전체 배열의 시작 주소 (값은 같지만 다른 자료형)
 - 일반 변수와 배열 비교

일반 변수	배열
<pre>int i=9, *ip = &i;</pre> <p>i : 변수 i에 저장된 값 &i : 변수 i의 주소</p>	<pre>int ar[5]={2, 3, 5, 7, -1};</pre> <p>ar[2] : 원소 ar[2]에 저장된 값 &ar[2] : 원소 ar[2]의 주소</p>
<p>ip : 변수 ip에 저장된 값 &ip : 변수 ip의 주소</p>	<p>ar : 0번 원소의 주소 &ar : (전체) 배열의 주소</p>



※ 배열 원소는 일반 변수와 동일하게 취급됨

3) 배열과 포인터

- 주소를 이용한 배열 참조

- 배열 이름은 주소를 의미하므로, **참조 연산자**와 함께 사용 가능

- ✓ `ar` : 0번 원소의 주소

- ✓ `*ar` : 0번 원소의 주소에 저장된 값, 즉, 0번 원소의 값을 의미

```
int ar[5]={2, 3, 5, 7, -1};  
  
printf("%p %d %d\n", ar, ar[0], *ar);
```

실행 결과

001E40B4 2 2

ar	0xB4	0xB8	0xBC	0xC0	0xC4
	2	3	5	7	-1
	[0]	[1]	[2]	[3]	[4]

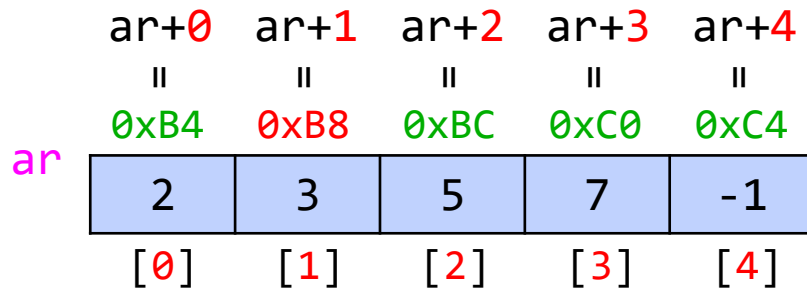
3) 배열과 포인터

- 배열 주소에 대한 증감 연산

- 배열 원소 하나의 크기 만큼 증가 or 감소 (int 배열의 경우: 4)
- $ar+i$: 배열 ar 의 i 번째 원소의 주소
- $*(ar+i)$: 배열 ar 의 i 번째 원소의 값, 즉, $ar[i]$

```
int ar[5]={2, 3, 5, 7, -1};
```

```
printf("%p %d %d\n", ar+1, ar[1], *(ar+1));
```



실행 결과

001E40B8 3 3

↑
 $ar+1$ 의 값은
001E40B5가 아님!!

3) 배열과 포인터

- [예제 9.4] char형 배열과 double형 배열을 선언하고, 다음을 출력하라.

```
char car[5]={'H','e','l','l','o'};  
double dar[5]={1.1, 2.2, 3.3, 4.4, 5.5};
```

- ✓ car, car[0], *car
- ✓ car+1, car[1], *(car+1)
- ✓ car+2, car[2], *(car+2)

**주소의 변화량을
주의해서 살펴보자**

- ✓ dar, dar[0], *dar
- ✓ dar+1, dar[1], *(dar+1)
- ✓ dar+2, dar[2], *(dar+2)

3) 배열과 포인터

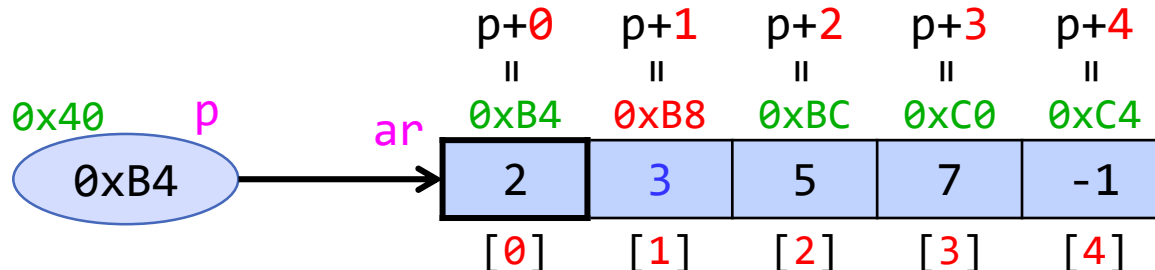
- 배열을 **포인터 변수**에 연결하여 사용하기
 - 배열 이름은 주소를 의미하므로, **포인터 변수에 대입 가능**
 - 포인터 변수에 대한 증감 연산
 - ✓ 포인터 변수가 **나타내는 자료형의 크기 단위로** 증가 or 감소

```
int ar[5]={2, 3, 5, 7, -1};  
int *p = ar;      ⇒ 포인터 변수에 ar 대입
```

```
printf("%p %d\n", p, *p);      ⇒ 0번 원소  
printf("%p %d\n", p+1, *(p+1)); ⇒ 1번 원소
```

실행 결과

001E40B4	2
001E40B8	3



3) 배열과 포인터

- 포인터 변수도 **배열의 첨자 형태**로 값을 참조 할 수 있다.

```
int ar[5]={2, 3, 5, 7, -1};  
int *p = ar;  
  
printf("%p %d %d\n", p, p[0], *p);           ⇒ 0번 원소  
printf("%p %d %d\n", p+1, p[1], *(p+1));     ⇒ 1번 원소
```

실행 결과

001E40B4	2	2
001E40B8	3	3

3) 배열과 포인터

- [예제 9.5] 다음과 같이 포인터 변수를 선언하고 값을 출력하라.

```
char car[5]={'H','e','l','l','o'}, *cp=car;  
double dar[5]={1.1, 2.2, 3.3, 4.4, 5.5}, *dp=dar;
```

- ✓ cp, cp[0], *cp
- ✓ cp+1, cp[1], *(cp+1)
- ✓ cp+2, cp[2], *(cp+2)

- ✓ dp, dp[0], *dp
- ✓ dp+1, dp[1], *(dp+1)
- ✓ dp+2, dp[2], *(dp+2)

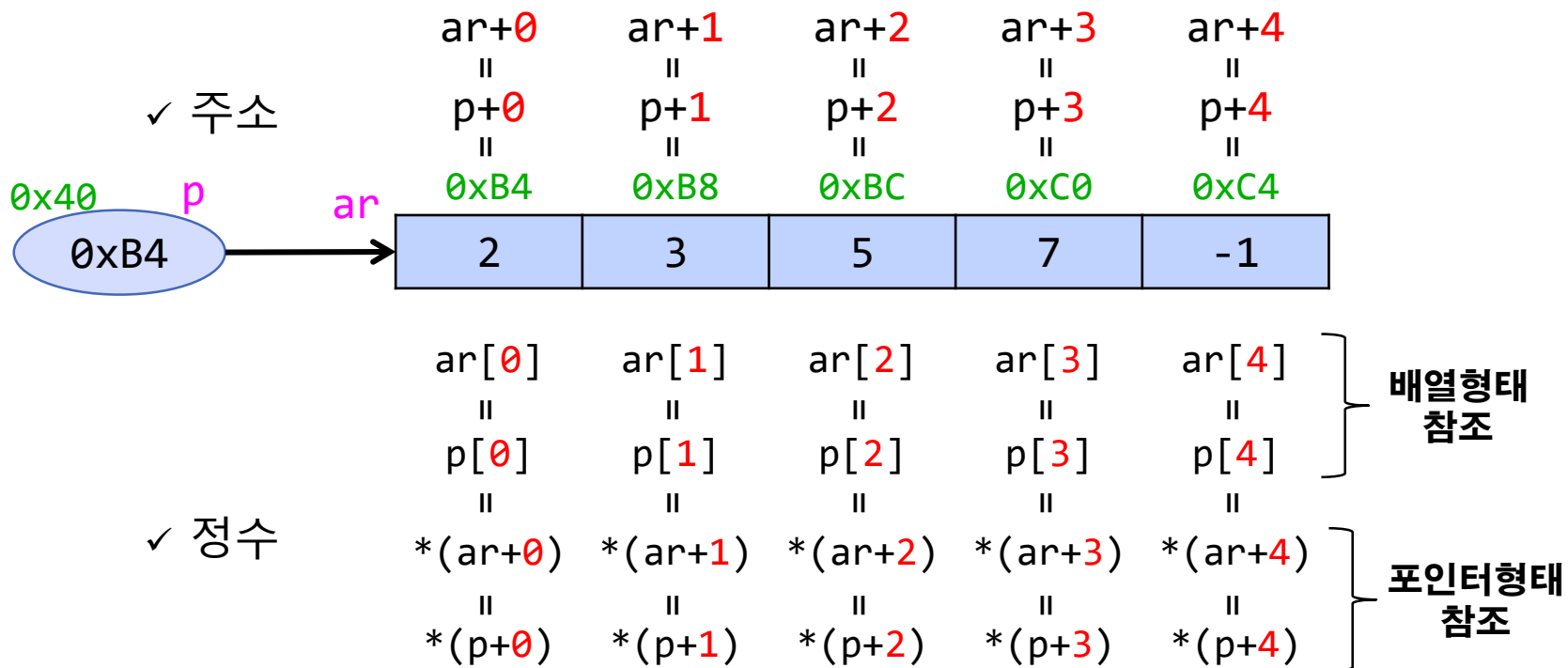
주소의 **변화량**을
주의해서 살펴보자

3) 배열과 포인터

- 배열과 포인터의 관계 정리

- 배열과 포인터는 동일한 형태로 사용 가능 (둘 다 주소이니까)

```
int ar[5], *p = ar;
```



3) 배열과 포인터

- 배열과 포인터 정리 (복잡해 보이지만 다음 두 가지만 기억하자)

```
int ar[5], *p = ar;
```

- 주소에 1을 더하면, 원소의 크기만큼 주소가 증가한다.
 - ✓ $ar + 3$, $p + 3$: ar 과 p 모두 주소
- 주소가 주어 졌을 때, 해당 주소에 저장된 원소(변수) 값은 다음 두 가지 형태로 참조할 수 있다.
 - ✓ $ar[3]$ 과 $p[3]$: 배열의 첨자 연산자 $[]$ 사용
 - ✓ $*(ar+3)$ 과 $*(p+3)$: 포인터의 참조 연산자 $*$ 사용

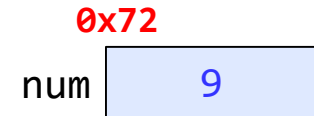
배열 이름이든 포인터 변수이든 주소를 의미하고,
따라서 참조 방식도 동일하다.

3) 배열과 포인터

배열 이름과 포인터 변수의 차이점

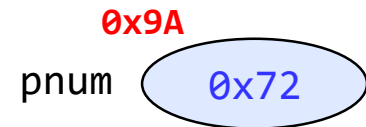
- `int num;`

- ✓ 변수 `num`에 저장된 값(정수)은 변경 가능
- ✓ 변수 `num`에 할당된 주소는 변경 불가



- `int *p;`

- ✓ 변수 `p`에 저장된 값(주소)은 변경 가능
- ✓ 변수 `p`에 할당된 주소는 변경 불가



- `int ar[5];`

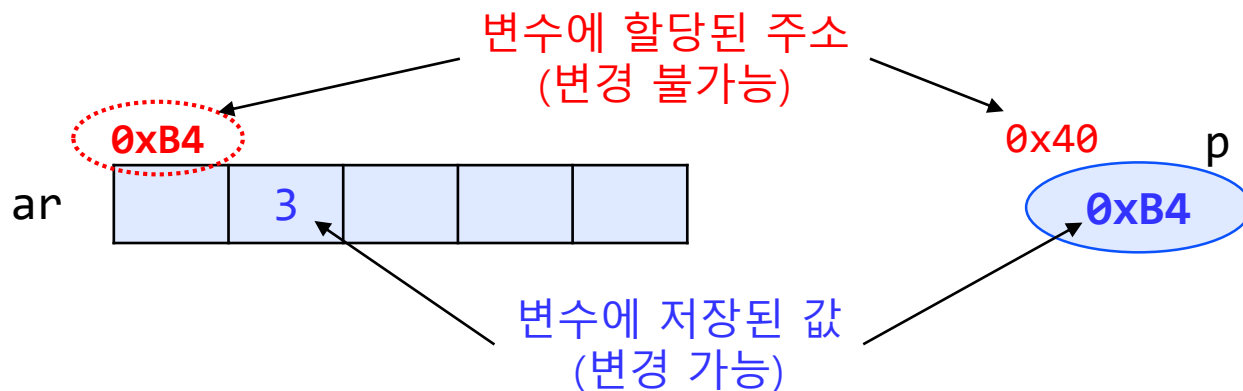
- ✓ 배열 `ar`에 저장된 값은 변경 가능
- ✓ 배열 `ar`에 할당된 주소는 변경 불가
 - ✓ 배열 이름은 포인터 상수로 변경 하지 못한다.
 - ✓ 대입문의 왼쪽에서 사용될 때 (l-value) 차이 발생



3) 배열과 포인터

- 배열 이름과 포인터 변수의 차이점

```
int num, *p, ar[5];  
  
p = &num;      // 가능  
++p;           // 가능  
&p = ar;       // 불가능 (컴파일 에러)  
  
ar = &num;      // 불가능 (컴파일 에러)  
++ar;          // 불가능  
&ar = &num;    // 불가능 (컴파일 에러)
```



3) 배열과 포인터

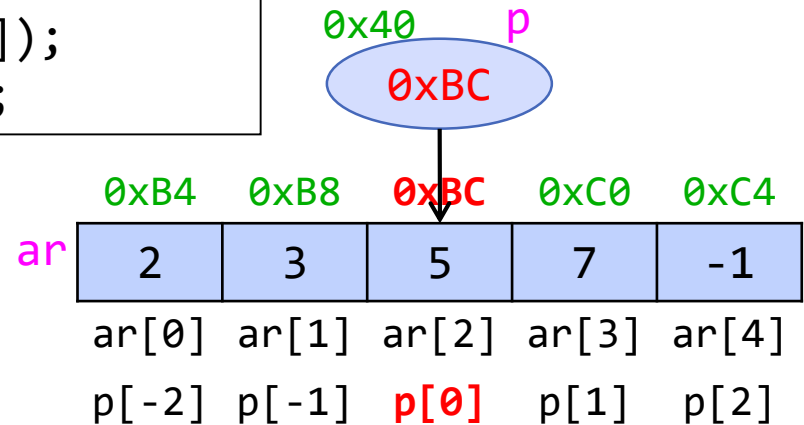
- 배열과 포인터 주의사항 1

- 포인터를 배열의 중간 원소에 연결시키는 것도 가능

```
int ar[5]={2, 3, 5, 7, -1};  
int *p = &ar[2];    // 2번 원소에 연결  
  
printf("%p %d\n", ar, ar[0]);  
printf("%p %d\n", p, p[0]);
```

실행 결과

001E40B4	2
001E40BC	5



- 포인터는 단지 자신이 가리키는 주소를 기준으로 배열처럼 쓰는 것일 뿐

3) 배열과 포인터

- 배열과 포인터 주의사항 2

- 포인터의 참조 연산자 사용시 괄호에 유의

- ✓ $\ast(\text{ar}+2) \rightarrow \text{ar}[2] \rightarrow 5$

- ✓ $\ast\text{ar} + 2 \rightarrow \ast(\text{ar}) + 2 \rightarrow \text{ar}[0]+2 \rightarrow 4$ (연산자 우선순위 때문)

ar	2	3	5	7	-1
	ar[0]	ar[1]	ar[2]	ar[3]	ar[4]

3) 배열과 포인터

- 배열과 포인터 주의사항 3

- 포인터 변수의 증감량은 가리키는 배열의 원소 크기가 아니라, 포인터 자신의 자료형에 의해 결정
 - ✓ 예) char * 형 포인터에 int 배열을 연결하면

```
int ar[5]={2, 3, 5, 7, -1}, i;  
char *p = (char *) ar;  
  
for( i=0; i < 5 ; ++i )  
    printf("%p, %d\n", p+i, *(p+i));
```

1씩 증가

```
001E40B4, 2  
001E40B5, 0  
001E40B6, 0  
001E40B7, 0  
001E40B8, 3
```

목차

- 1) 포인터 개요
- 2) 포인터 선언과 사용
- 3) 배열과 포인터
- 4) **포인터 연산**
- 5) 포인터 인자와 주소 반환
- 6) 포인터 배열
- 7) 다중 포인터

4) 포인터 연산

- 주소에 정수를 더하거나 빼기

- ++, --, +=, -= 와 같이 덧셈, 뺄셈에 대한 연산자는 모두 가능

```
int ar[5] = {2, 3, 5, 7, -1}, *p = ar;
int i = 4;

printf("%p %d\n", p+2, *(p+2));    ⇨ &ar[2], ar[2]
printf("%p %d\n", p+i, *(p+i));    ⇨ ar[4]
printf("%p %d\n", ++p, *p);        ⇨ ar[1]
```

실행 결과

```
001E40BC 5
001E40C4 -1
001E40B8 3
```

4) 포인터 연산

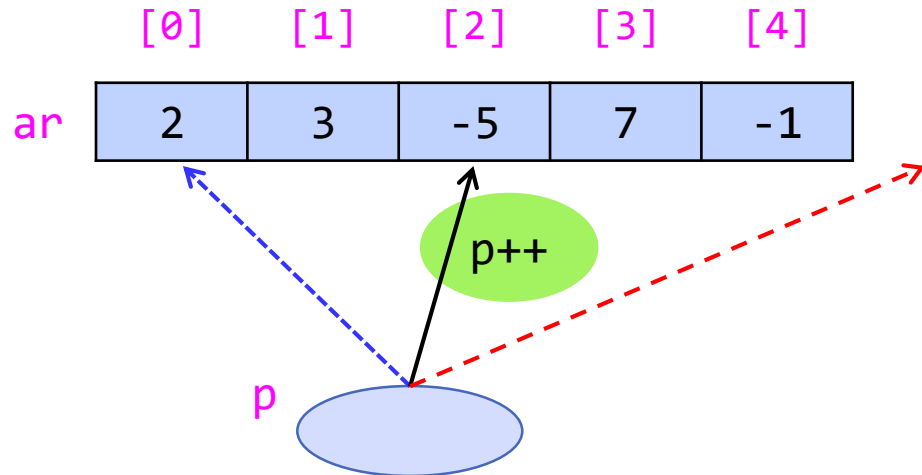
- [예제 9.6] 다음에 해당하는 문장을 차례로 작성하고, p1과 p2의 값이 얼마나 증가하는지 확인해보자. (배열과 포인터의 주의사항 3 참고)
 - ① int 포인터 p1을 선언하고, NULL로 초기화
 - ② char 포인터 p2를 선언하고, NULL로 초기화
 - ③ p1과 p2를 출력 (즉, p1과 p2에 저장된 값 출력)
 - ④ p1과 p2를 1만큼 증가 (++ 연산자 사용)
 - ⑤ p1과 p2를 출력
 - ⑥ p1과 p2를 2만큼 증가 (+= 연산자 사용)
 - ⑦ p1과 p2를 출력

4) 포인터 연산

- [예제 9.7] 포인터 연산을 이용하여 배열 전체 훑어보기
 - ① 정수 배열 `ar[5]`를 선언하고 {2, 3, 5, 7, -1}로 초기화
 - ② 정수 변수 `i`와 정수 포인터 `p` 선언 후, `p`에 배열 이름 `ar` 연결
 - ③ for문: 다음을 5회 반복 (`i`는 반복 제어 변수)
 - ✓ `p`가 가리키는 변수의 값 출력
 - ✓ `p`의 값 1만큼 증가

실행결과

2
3
5
7
-1



4) 포인터 연산

▪ 주소 비교하기

- 비교 연산자(==, !=, <, >, >=, <=) 사용 가능

```
int ar[5] = {2, 3, 5, 7, -1}, *p1 = &ar[1], *p2 = &ar[4];
```

```
printf("%p %p\n", p1, p2);    ⇒ ar[1]과 ar[4]의 주소
```

```
printf("%d\n", p1 < p2);      ⇒ ar[1]과 ar[4]의 주소 값 비교
```

```
printf("%d\n", *p1 < *p2);    ⇒ ar[1]과 ar[4]의 원소 값 비교
```

실행결과

```
001E40B8 001E40C4
```

```
1    ⇒ p1과 p2에 저장된 값 비교
```

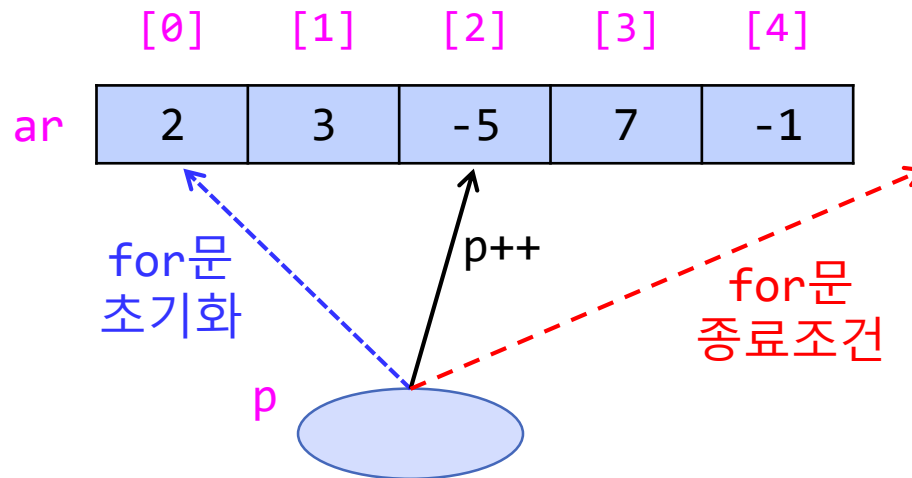
```
0    ⇒ p1과 p2가 가리키는 변수의 값 비교
```

4) 포인터 연산

- 주소 비교를 이용하여 **배열 훑어보기**

(참고) 포인터 학습을 위한 연습용 코드,
보통 배열은 반복 제어 변수 사용

```
int ar[5]={2, 3, 5, 7, -1}, *p;  
for( p = ar ; p < &ar[5] ; p++ )  
    printf("%d\n", *p);
```



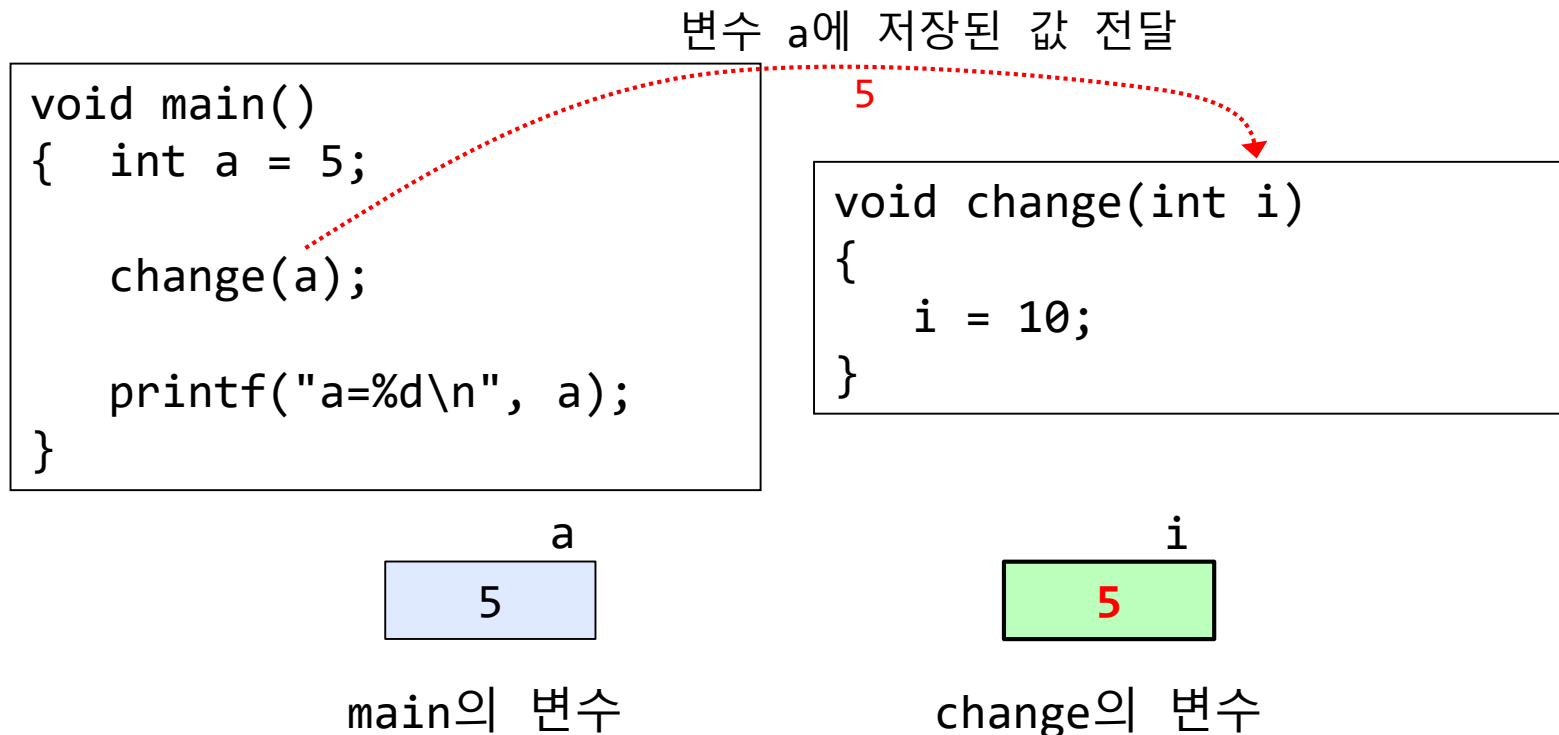
목차

- 1) 포인터 개요
- 2) 포인터 선언과 사용
- 3) 배열과 포인터
- 4) 포인터 연산
- 5) **포인터 인자와 주소 반환**
- 6) 포인터 배열
- 7) 다중 포인터

5) 포인터 인자와 주소 반환

- 함수 수행 과정 (복습) - 정수 인자

- A. 함수 시작(호출): **형식 인자**(변수)에 공간이 할당되고, 각 인자에 전달된 **정수 값이 대입됨**.



change 함수 시작 시 메모리 그림

5) 포인터 인자와 주소 반환

- B. 함수 본체 수행: 지역변수 i에 10 대입
- C. 함수 종료: 함수의 지역변수(인자 포함)가 없어짐 (할당된 메모리 공간 반환)

```
void main()
{  int a = 5;

    change(a);

    printf("a=%d\n", a);
}
```

결과:

a=5

a
5
main의 변수

```
void change(int i)
{
    i = 10;
}
```

~~i~~
10
change의 변수
⇒ 함수 종료 시
공간 해제됨

change 함수 종료 시 메모리 그림

5) 포인터 인자와 주소 반환

- 함수 인자가 정수가 아니라 **주소**라면?
 - 코드에서 변경되는 부분
 - ✓ 주소를 저장하기 위해 함수의 형식 인수는 **포인터**로 선언
 - ✓ 포인터 변수를 이용한 **간접 참조**

```
void main()
{  int a = 5;

    change(&a);

    printf("a=%d\n", a);
}
```

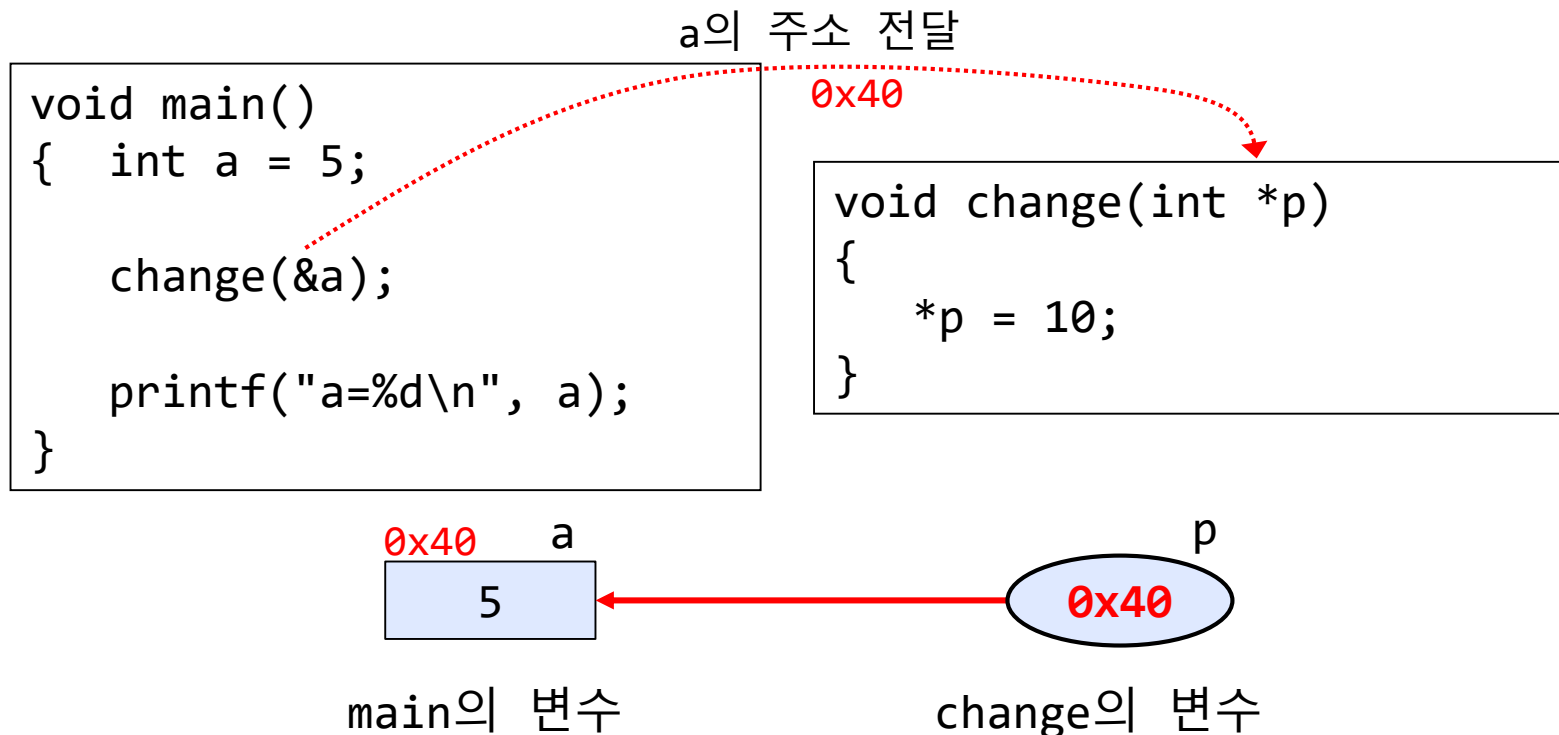
```
void change(int *p)
{
    *p = 10;
}
```

- 위 함수의 수행 과정은? 다음 슬라이드

5) 포인터 인자와 주소 반환

- 함수 수행과정

- A. 함수 시작(호출): **형식 인자**(변수)에 공간이 할당되고, 각 인자에 전달된 **주소가 대입됨**.



change 함수 시작 시 메모리 그림

5) 포인터 인자와 주소 반환

- B. 함수 본체 수행: p가 가리키는 변수에 10 대입
- C. 함수 종료: 함수의 지역변수(인자 포함)가 없어짐 (할당된 메모리 공간 반환)

```
void main()
{  int a = 5;

    change(&a);

    printf("a=%d\n", a);
}
```

```
void change(int *p)
{
    *p = 10;
}
```

결과:

a=10

0x40 a
10
main의 변수

~~p~~
~~0x40~~
change의 변수

함수 종료 시
공간 해제됨

change 함수 종료 시 메모리 그림

5) 포인터 인자와 주소 반환

▪ 함수 인자 비교

- 정수 값이나 문자 등을 인자로 함수를 호출하는 것을
'값에 의한 호출(call-by-value)'이라 함
 - ✓ 호출하는 함수의 변수에 영향을 못 미침
- 주소를 인자로 함수를 호출하는 것을
'주소에 의한 호출(call-by-reference)'이라 함
 - ✓ 간접 참조로 인해 호출하는 함수의 변수 값을 변경시킬 수 있음
- 하지만, 두 호출 방식의 함수의 호출 과정(인자 전달 및 제어 흐름)은 **완전히 동일**
 - ✓ 값이 전달되느냐, 주소가 전달되느냐에 따라오는 부수적인 효과일 뿐

5) 포인터 인자와 주소 반환

- 추가 예시

- 함수의 실 인자가 포인터 변수인 경우

```
void main()
{   int a = 5;
    int *pa = &a;

    change(pa);

    printf("a=%d\n", a);
}
```

```
void change(int *p)
{
    *p = 10;
}
```

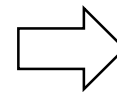
✓ 메모리 그림을 그리면서 수행과정을 따져보자

5) 포인터 인자와 주소 반환

- 포인터 인자 활용 예제: 두 변수의 값을 교환하는 함수
 - 아래 swap 함수를 이용하면 main의 값이 바뀌는가? **NO!!**

```
void swap(int x, int y){  
    int tmp = x;  
    x = y;  
    y = tmp;  
    printf("%d %d\n", x, y);  
}
```

```
void main(){  
    int x = 10, y = 20;  
  
    swap(x, y);  
  
    printf("%d %d\n", x, y);  
}
```

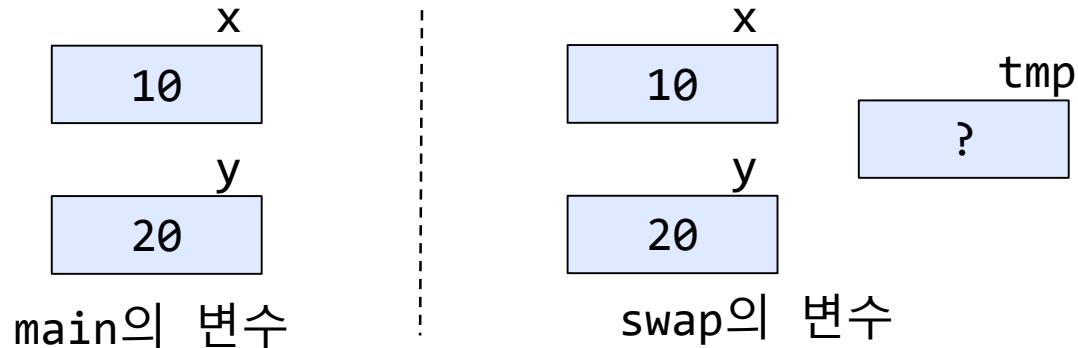


실행 결과

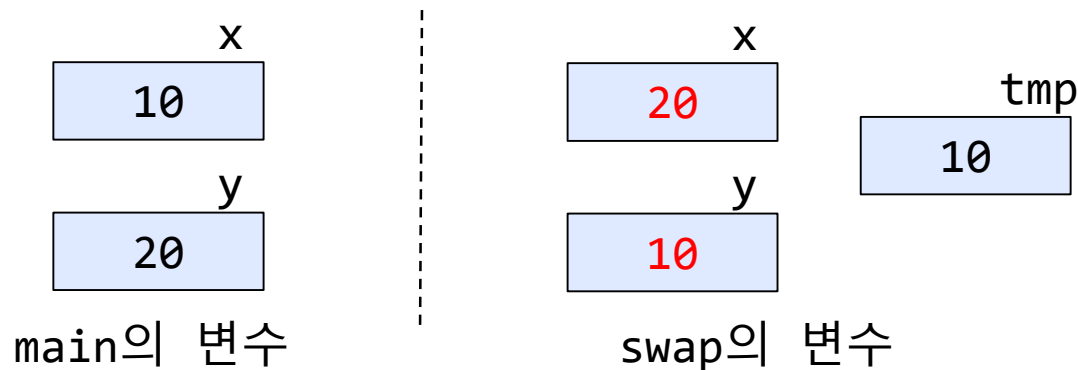
20	10	(swap)
10	20	(main)

5) 포인터 인자와 주소 반환

- 왜 안 바뀌는 지 메모리 그림을 그려 확인해보자.



swap 함수 시작 시 메모리 그림

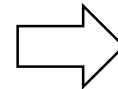


swap 함수 종료 직전 메모리 그림

5) 포인터 인자와 주소 반환

- swap함수에서 **포인터를 인자**로 사용하면?
 - ✓ 함수의 인자는 정수 포인터
 - ✓ 호출 시 주소 전달

```
void swap(int *px, int *py){  
    int tmp = *px;  
    *px = *py;  
    *py = tmp;  
    printf("%d %d\n", *px, *py);  
}  
  
void main(){  
    int x = 10, y = 20;  
  
    swap(&x, &y);  
  
    printf("%d %d\n", x, y);  
}
```

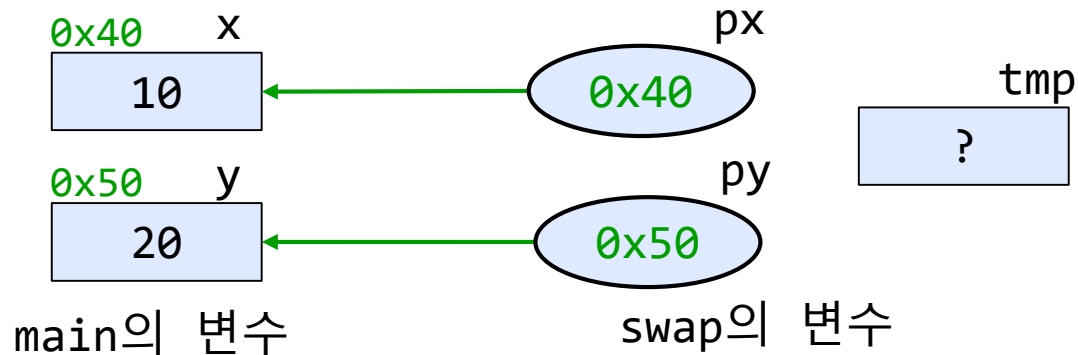


실행 결과

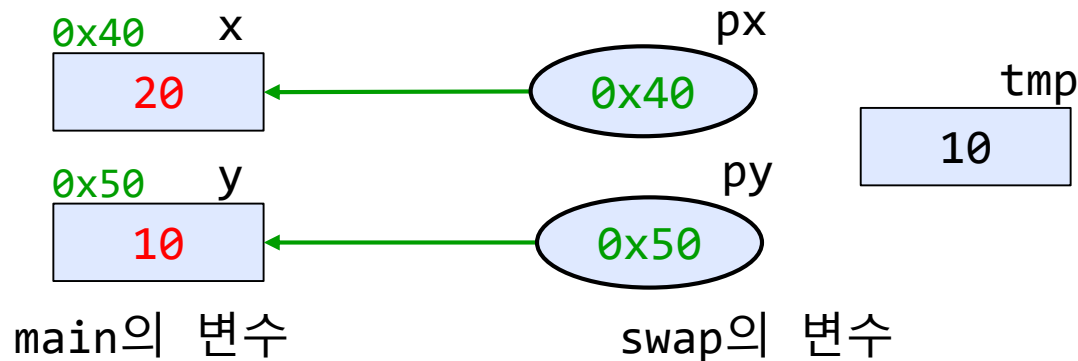
20	10	(swap)
20	10	(main)

5) 포인터 인자와 주소 반환

- main 함수의 값이 왜 바뀌는 지 메모리 그림을 그려 확인해보자.



swap 함수 시작 시 메모리 그림



swap 함수 종료 직전 메모리 그림

5) 포인터 인자와 주소 반환

배열 인자 다시 보기

- 아래 코드에서 init 함수에서 배열 값을 변경하면, main의 배열 값도 변경된다. (함수 단원에서 학습했음)
- 왜 그럴까? 배열의 시작 **주소**가 인자로 전달되기 때문

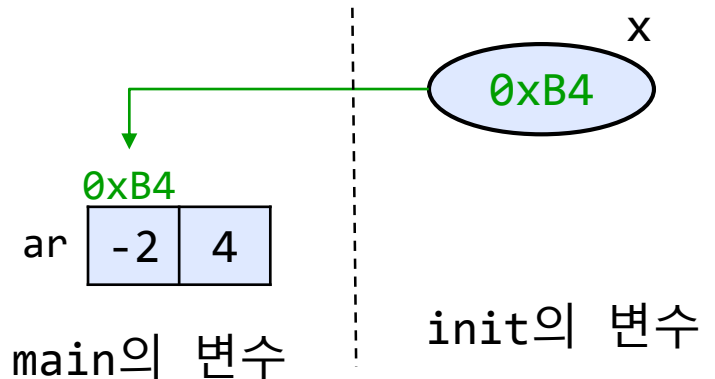
```
void init(int x[]){
    x[0] = x[1] = 0;
}
void main(){
    int ar[2]={-2,4};

    init(ar);

    printf("%d %d",ar[0],ar[1]);
}
```

실행 결과

0 0



5) 포인터 인자와 주소 반환

- init 함수의 형식 인자 `int x[]`은 **포인터 (배열 아님에 주의)**

```
void init(int x[]){  
    ...  
}
```

=

```
void init(int *x){  
    ...  
}
```

- 위 두 함수는 문법적으로 완전히 동일
- 배열과 포인터 중 의미를 두고자 하는 형태로 사용
 - ✓ 보통 함수 본체에서 배열 형태로 사용하는 경우,
배열 형태의 인자 사용(왼쪽 형태)

5) 포인터 인자와 주소 반환

- scanf와 printf의 인자

- scanf()에서 변수 앞에 &(주소 연산자)를 붙이는 이유

```
int x;  
scanf("%d", &x);
```

- ✓ 사용자로 부터 입력 받은 값을 변수 x에 저장해야 한다.
- ✓ scanf 함수에서 호출한 함수의 변수 x의 값을 바꾸려면, 주소를 전달

- printf()는?

```
int x = 0;  
printf("%d", x);
```

- ✓ 출력 시는 x의 값을 넘겨주면 충분

5) 포인터 인자와 주소 반환

- scanf()에서 항상 **&**(주소 연산자)를 붙여야 하는가?
 - ✓ 주소 값이 인자로 전달되면 됨

```
int x[5], *p = &x[1];

scanf("%d", &x[0]);
scanf("%d", p);
scanf("%d", p+1);

printf("%d %d %d\n", x[0], x[1], x[2]);
printf("%p %p %p\n", &x[0], &x[1], &x[2]);
```

입력 예시

1 -4 9

출력 예시

1 -4 9

00B6FB60 00B6FB64 00B6FB68

5) 포인터 인자와 주소 반환

▪ 주소를 반환하는 함수

- 주소도 값이므로 함수의 반환값으로 사용될 수 있다.
- 변수와 마찬가지로, 함수 이름 앞에 ***** (참조연산자) 을 붙여 주소를 반환함을 표시

```
void main(){
    int ar[5]={2,1,3,0,4};
    int *p1;

    p1 = next_addr(&ar[1]);
    printf("%d", *p1);
}
```

```
int *next_addr(int *p)
{
    return p+1;
}
```

실행 결과

3

5) 포인터 인자와 주소 반환

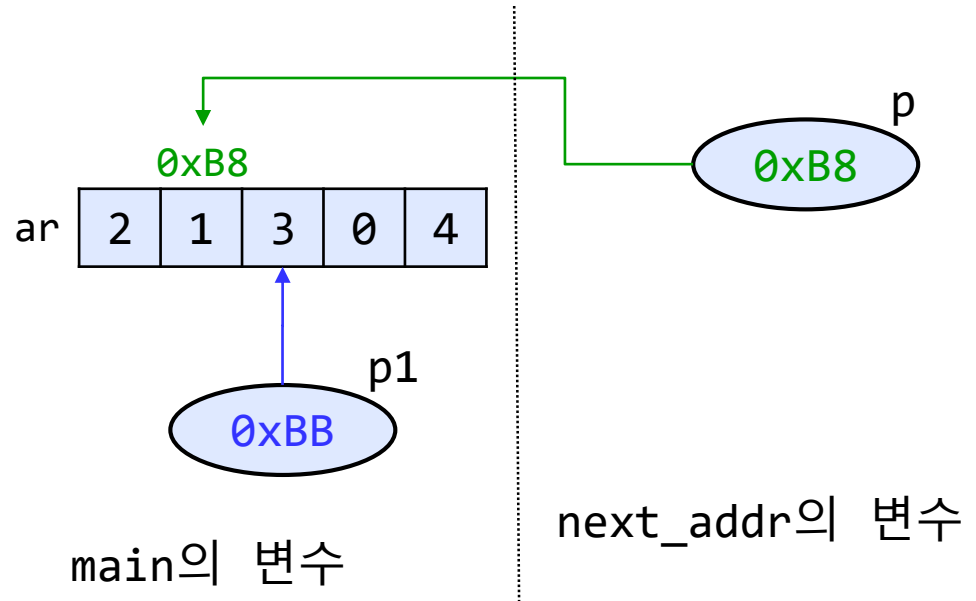
- 앞의 코드에 대해 메모리 그림을 그리면?

```
int *next_addr(int *p)
{
    return p+1;
}

void main(){
    int ar[5]={2,1,3,0,4};
    int *p1;

    p1 = next_addr(&ar[1]);

    printf("%d",*p1);
}
```



next_addr 함수 종료 직전 메모리 그림

5) 포인터 인자와 주소 반환

- [예제 9.8] 두 정수 변수의 **주소**를 인자로 받아, 값이 작은 변수의 **주소**를 반환하는 함수를 작성해보자.
 - 변수에 저장된 값은 다르다고 가정
 - main 함수는 아래의 코드 사용

```
void main(){
    int ar[5]={2,1,3,0,4};
    int *p1;

    p1 = smaller(&ar[1], &ar[3]);

    printf("%d",*p1);
}
```

```
?    smaller (    ?    )
{
    ?
}
```

실행 결과

0

목차

- 1) 포인터 개요
- 2) 포인터 선언과 사용
- 3) 배열과 포인터
- 4) 포인터 연산
- 5) 포인터 인자와 주소 반환
- 6) 포인터 배열**
- 7) 다중 포인터

6) 포인터 배열

- 포인터 배열: 포인터 변수들의 묶음
- 포인터 배열 선언
 - 포인터 선언 + 배열 선언

```
void main(){
    int a=1, b=2, c=3, i;
    int *pi[3]; // 포인터 배열 선언

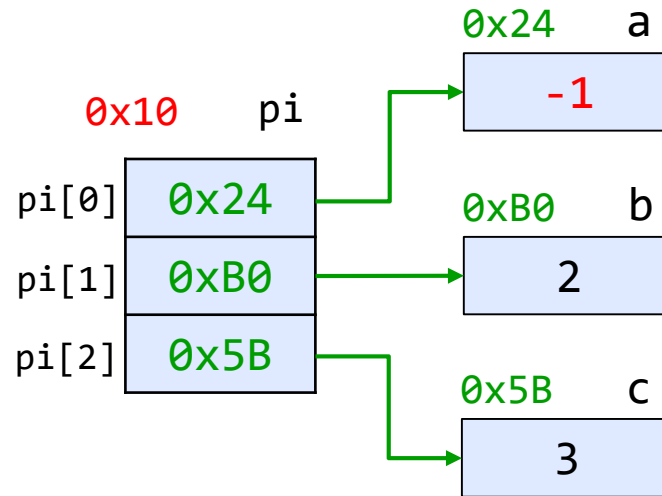
    pi[0] = &a; // 배열 원소는 int 포인터 변수
    pi[1] = &b; pi[2] = &c;

    *pi[0] = -1; // *pi[0]은 *(pi[0]) ? (*pi)[0] ?

    for( i=0; i < 3 ; ++i )
        printf("%p %p %d\n", &pi[i], pi[i], *pi[i]);
}
```

6) 포인터 배열

- 메모리 그림

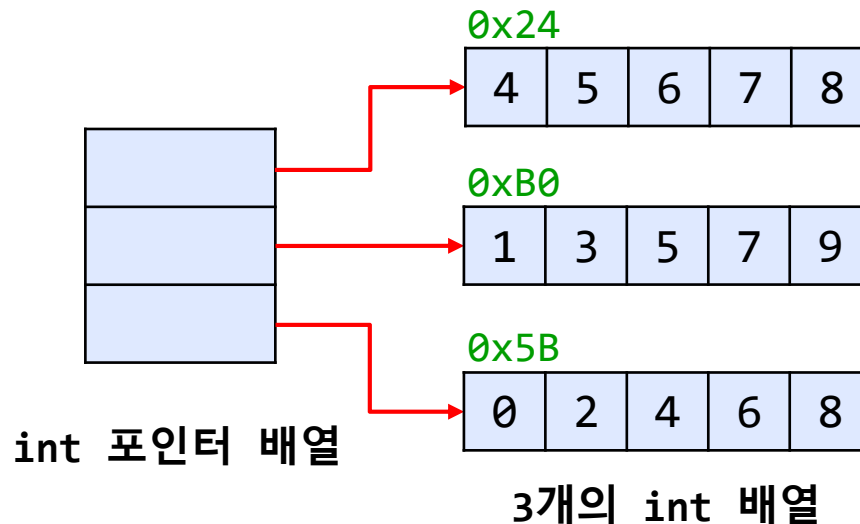


실행 결과

006FFD10	006FFD24	-1
006FFD14	006FFDB0	2
006FFD18	006FFD5B	3

6) 포인터 배열

- [예제 9.9] 다음 프로그램을 작성하시오.
 - 3개의 int 배열을 선언 (배열의 크기는 모두 5)하고 아래 그림과 같이 초기화
 - 1개의 int **포인터 배열** 선언 (배열의 크기는 3)
 - 아래 그림과 같이 **포인터 배열의 각 원소**를 각 int 배열에 연결
 - (뒷장에 계속)

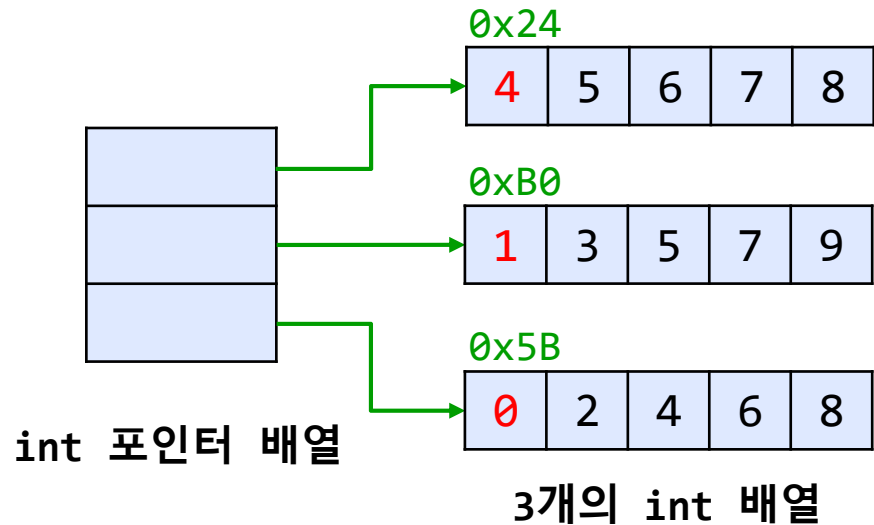


5) 포인터 배열

- int 포인터 배열을 이용하여 각 int 배열의 0번 원소의 주소와 값을 출력하시오. (앞 예제 참조)
 - ✓ 단, int 배열의 이름은 사용 금지

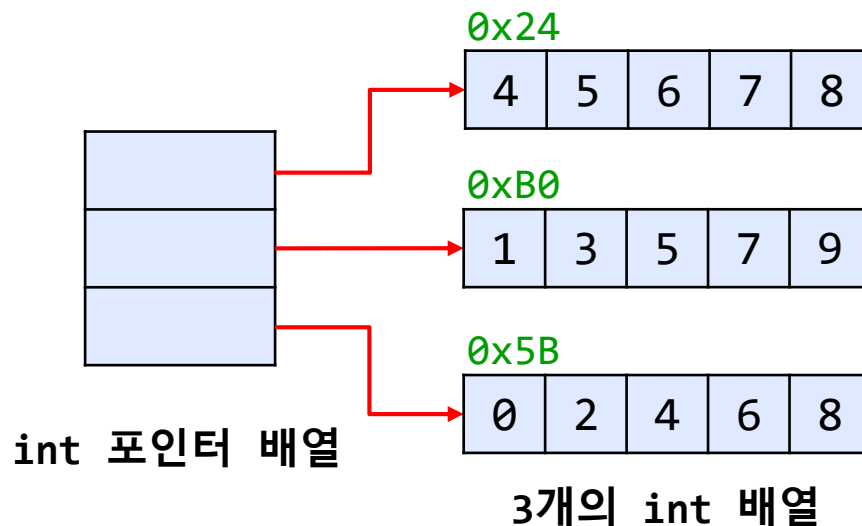
실행 결과 (예시)

006FFD24	4
006FFDB0	1
006FFD5B	0



6) 포인터 배열

- [예제 9.10] 이전 실습에서 int 포인터 배열을 이용하여 int 배열의 모든 원소의 값을 출력하시오.
 - 원소 값 출력 코드에서 int 배열의 이름 사용 금지
 - (힌트!!) 2중 반복문, 9.3절 배열과 포인터의 관계



실행 결과

4	5	6	7	8
1	3	5	7	9
0	2	4	6	8

6) 포인터 배열

- 코드1 (임시 포인터 변수 x 이용)

```
void main(){
    ...           // int 배열 선언과 초기화 (생략)
    int *pi[3];    // int 포인터 배열 선언

    ...           // pi 배열에 int 배열 연결 (생략)

    for( i=0 ; i<3 ; ++i ){
        x = pi[i]; // 임시 변수 x에 pi 배열의 원소 값 대입

        for(j=0; j<5; j++)
            printf(" %d", x[j]); // 포인터를 배열처럼 사용
        printf("\n");
    }
}
```

6) 포인터 배열

- 코드2 (2차원 배열 형태 사용) : 일반적인 형태
 - ✓ 임시변수 x 사용 안 함 → 대신 pi[i] 사용

```
void main(){
    ...           // int 배열 선언과 초기화 (생략)
    int *pi[3];    // int 포인터 배열 선언

    ...           // pi 배열에 int 배열 연결 (생략)

    for( i=0 ; i<3 ; ++i ){
        x = pi[i]; // 임시 변수 x에 pi 배열의 원소 값 대입

        for(j=0; j<5; j++)
            printf(" %d", pi[i][j]); // 2차원 배열처럼
        printf("\n");
    }
}
```

목차

- 1) 포인터 개요
- 2) 포인터 선언과 사용
- 3) 배열과 포인터
- 4) 포인터 연산
- 5) 포인터 인자와 주소 반환
- 6) 포인터 배열
- 7) **다중 포인터**

7) 다중 포인터

- 이중 포인터

- 'int 포인터 변수의 주소'를 저장하는 변수? 가능
- 'int 포인터 변수의 주소'를 저장하는 변수의 자료형은? (**int ****)

- 선언

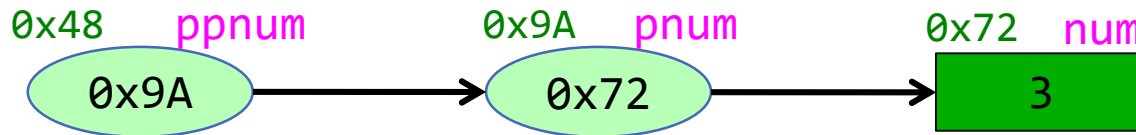
- 참조 연산자를 두 개 사용

```
int num=3, *pnum, **ppnum;
```

```
pnum = &num;      // 변수 num의 주소가 pnum에 저장됨
```

```
ppnum = &pnum;    // 변수 pnum의 주소가 ppnum에 저장됨
```

```
printf("%p %p %d\n",ppnum, pnum, num);
```



7) 다중 포인터

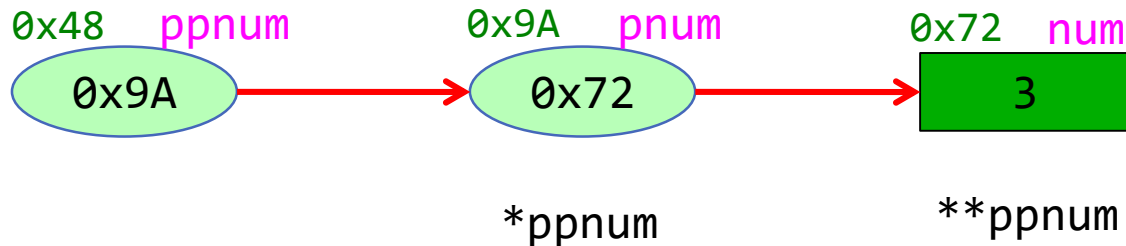
■ 이중 포인터의 참조 연산

- 참조연산자를 **한** 개 사용하면 간접 참조를 **한 번** (*****)
- 참조연산자를 **두** 개 사용하면 간접 참조를 **두 번** (******)

```
int num = 3, *pnum = &num, **ppnum = &pnum;  
printf("%p %p %d\n",ppnum, *ppnum, **ppnum);
```

결과:

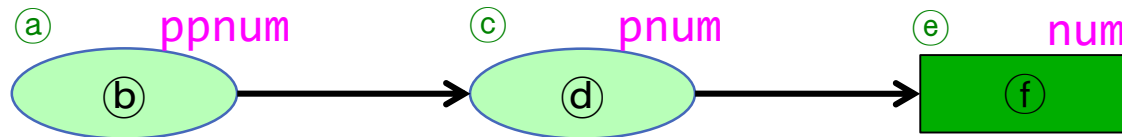
001EA09A 001EA072 3



7) 다중 포인터

- [예제 9.11] 다음 프로그램의 각 출력이 아래 메모리 그림에서 어느 부분을 출력하는 건지 생각해보고, 프로그램을 실행시켜 확인해보자.

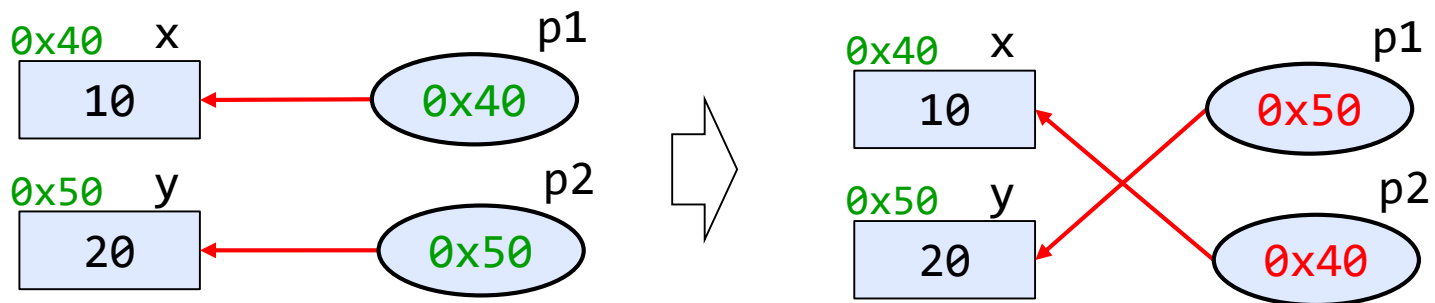
```
int num=3, *pnum = &num, **ppnum = &pnum;  
  
printf("%p %d\n", &num, num);  
printf("%p %p %d\n", &pnum, pnum, *pnum);  
printf("%p %p %p %d\n", &ppnum, ppnum, *ppnum, **ppnum);
```



7) 다중 포인터 (심화 내용)

- int 포인터 변수의 값을 서로 교환하는 프로그램
 - ✓ p1이 y를 가리키고, p2가 x를 가리키도록 변경

```
void main() {  
    int x = 10, y = 20;  
    int *p1 = &x, *p2 = &y, *ptmp;  
  
    ptmp = p1;           // p1 <-> p2 교환  
    p1 = p2;  
    p2 = ptmp;  
  
    printf("%d %d", *p1, *p2);  
}
```



7) 다중 포인터 (심화 내용)

- 교환하는 부분을 아래와 같이 함수로 만들어 보자.
 - ✓ 잘 바뀌는가? 함수에 포인터를 넘겨주었는데 왜 안 바뀔까?
 - ✓ 이유를 파악하기 위해 메모리 그림을 그려서 확인해보라.

```
void swap(int *p1, int *p2){
    int *ptmp = p1;
    p1 = p2;
    p2 = ptmp;
}

void main() {
    int x = 10, y = 20;
    int *p1 = &x, *p2 = &y;

    swap( p1, p2);    // p1과 p2의 저장 값 전달

    printf("%d %d", *p1, *p2);
}
```

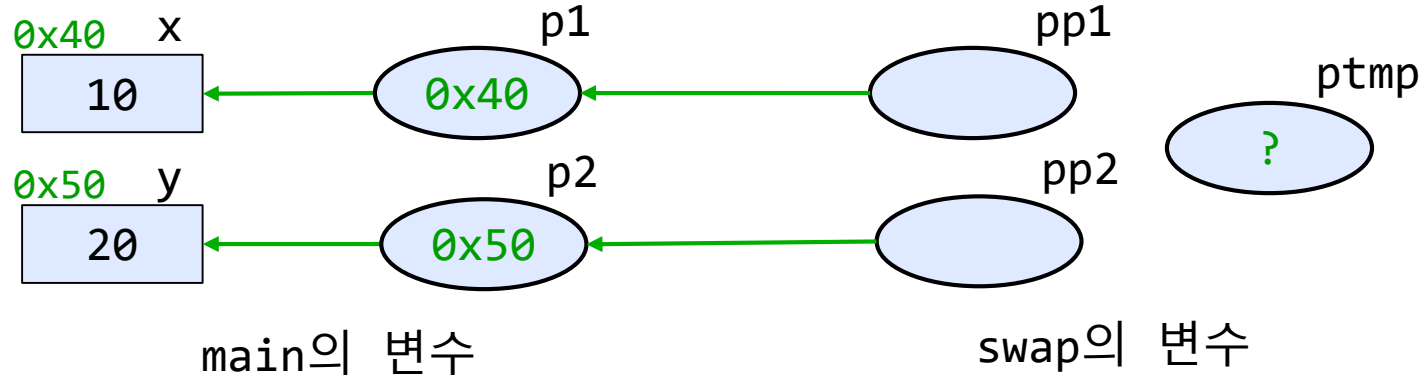
7) 다중 포인터 (심화 내용)

- 이중 포인터를 swap 함수의 인자로 사용하도록 수정
 - ✓ 교환하고자 하는 변수의 주소를 인자로 전달
 - ✓ p1, p2가 (int *) 형이므로 주소는 (int **) 형

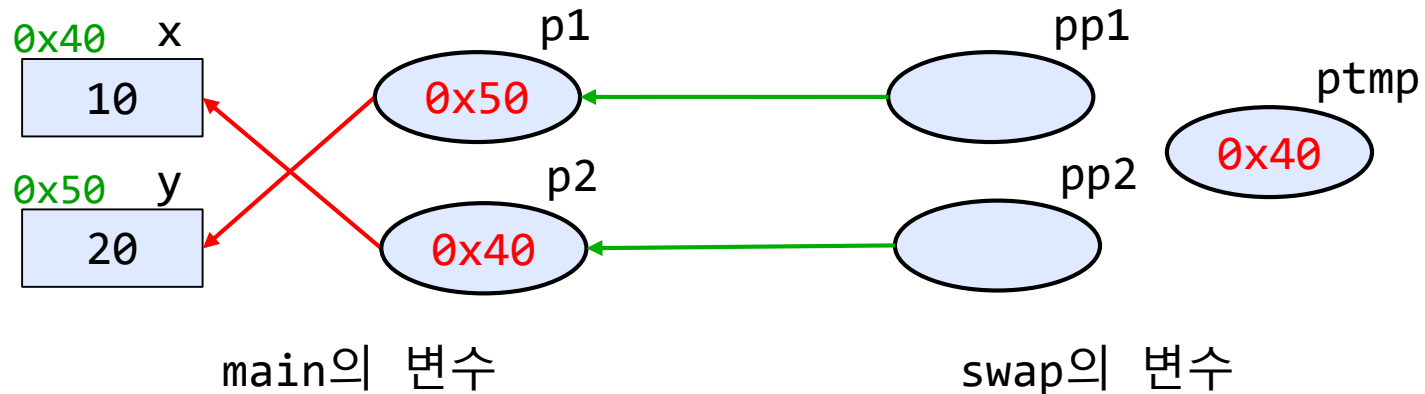
```
void swap(int **pp1, int **pp2){  
    int *ptmp = *pp1;  
    *pp1 = *pp2;  
    *pp2 = ptmp;  
}  
  
void main() {  
    int x = 10, y = 20;  
    int *p1 = &x, *p2 = &y;  
  
    swap( &p1, &p2);    // p1과 p2의 주소 전달  
  
    printf("%d %d", *p1, *p2);  
}
```

7) 다중 포인터 (심화 내용)

- main 함수의 값이 왜 바뀌는 지 메모리 그림을 그려 확인해보자.



swap 함수 시작 시 메모리 그림



swap 함수 종료 직전 메모리 그림

7) 다중 포인터 (심화 내용)

▪ 포인터배열과 이중포인터

- int 포인터 배열의 이름은 어떤 값과 자료형을 가질까?
 - ✓ 배열의 이름은 0번 원소의 주소와 동일한 값과 자료형을 가짐
 - ✓ 0번 원소의 자료형은 (int *) → 이 원소의 주소의 자료형은 (int **)

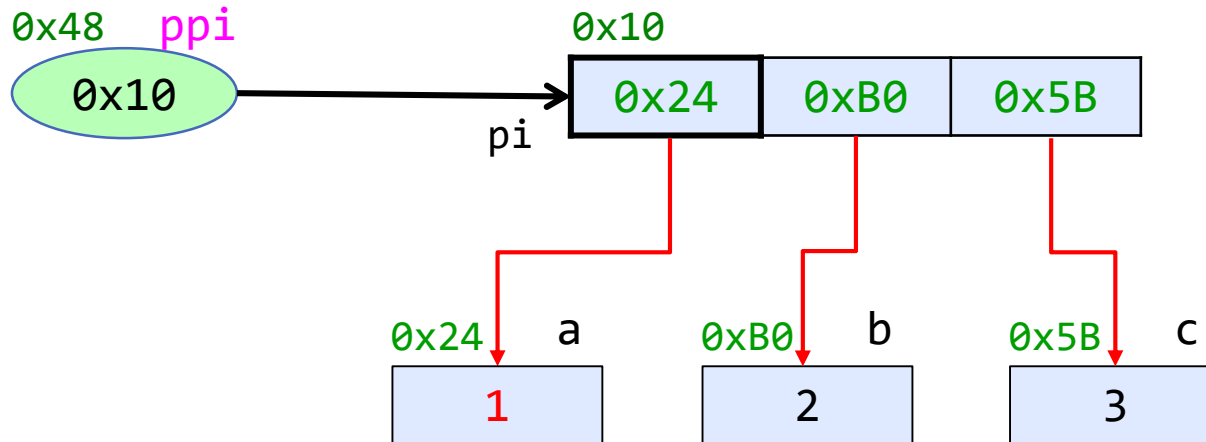
```
void main(){
    int a=1, b=2, c=3;
    int *pi[3] = {&a, &b, &c};    // int포인터 배열
    int **ppi;

    ppi = pi;    // 배열 이름 pi 는 &pi[0]와 동일

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

7) 다중 포인터 (심화 내용)

- 메모리 그림



7) 다중 포인터 (심화 내용)

- [예제 9.12] 앞 슬라이드의 프로그램에서 변수 a, b, c의 값을 **포인터 변수 ppi만을 이용하여** 출력하라.
 - 9.6절 포인터 배열의 코드 참고

7) 다중 포인터 (심화 내용)

- 이중 포인터와 비슷하게, 삼중, 사중, .. 포인터도 가능
 - 주의!! 각 포인터의 자료형은 모두 다르다.

```
int i = 10, *pi, **ppi, ***pppi, ****ppppi;
```

```
pi = &i          //(int)형인 i의 주소는 (int *)형
```

```
ppi = &pi;       //(int *)형인 pi의 주소는 (int **)형
```

```
pppi = &ppi;     //(int **)형인 ppi의 주소는 (int ***)형
```

```
ppppi = &pppi;  //(int ***)형인 pppi의 주소는 (int ****)형
```

```
printf("%d %d %d %d %d\n", ****ppppi, ***pppi, **ppi, *pi, i);
```

