



C 프로그래밍 및 실습

C Programming

이지민



# CHAP 08 포인터

# 학습목표

포인터란 무엇인지 알아보고, 포인터를 선언하고 사용하는 방법을 알아본다.

포인터를 사용할 때 주의사항을 알아본다.

포인터 연산에 대하여 알아본다.

배열과 포인터의 관계에 대하여 알아본다.

문자열 상수와 문자열 포인터에 대하여 알아본다.

# 목차



## 포인터의 기본

- 포인터란
- 포인터 사용 시 주의사항



## 배열과 포인터의 관계

- 포인터의 연산
- 포인터로서의 배열
- 배열의 원소를 가리키는 포인터
- 배열과 포인터의 차이점



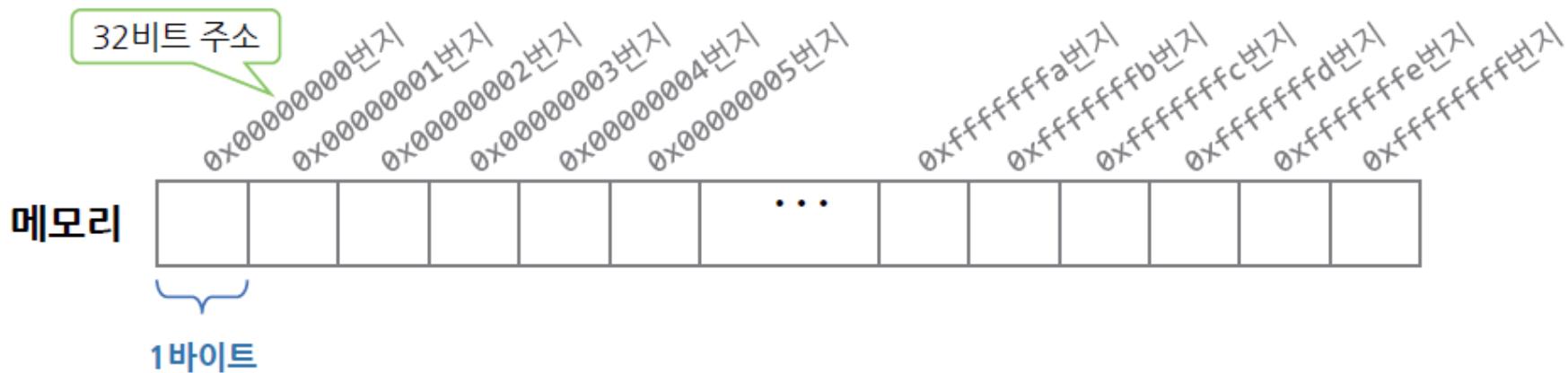
## 포인터와 문자열

- 문자열 상수
- 문자열 포인터
- const 포인터

# 포인터의 기본\_포인터란?

## ❖ 포인터의 개념(1/2)

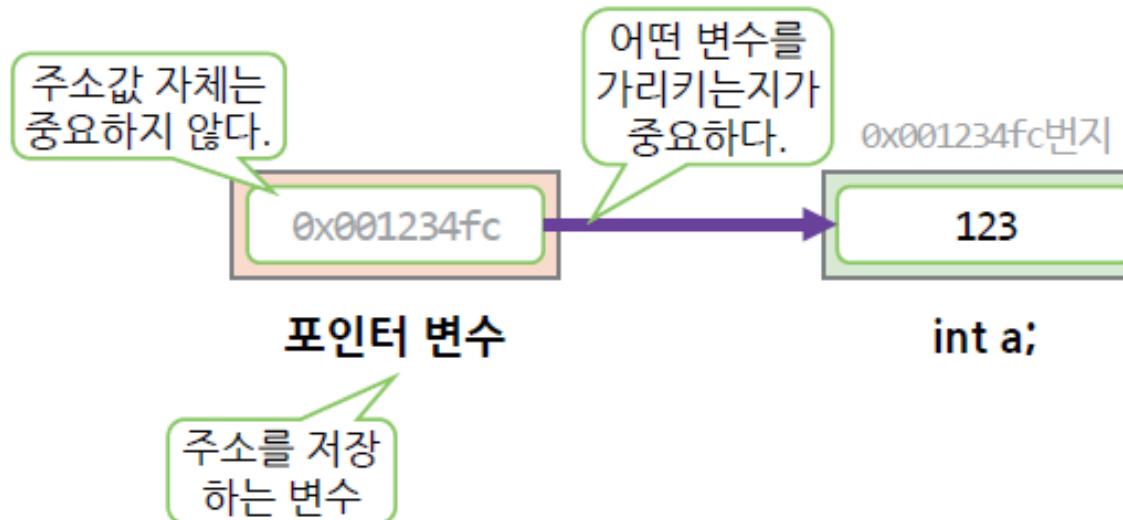
- 주소(**address**)를 저장하는 변수
- 메모리는 연속된 바이트의 모임이다.
  - 각각의 바이트를 구분하기 위해서 주소(번지)를 사용한다.
- 메모리 주소 공간(**address space**)
  - 메모리 주소가 가질 수 있는 범위
  - 0x00000000번지~0xffffffff번지 사이의 값 (4바이트 크기의 메모리 주소인 경우)



# 포인터의 기본\_포인터란?

## ❖ 포인터의 개념(2/2)

- 주소값 자체가 중요한 게 아니라 포인터가 어떤 변수를 가리키는지가 중요하다.
- 포인터는 다른 변수를 가리킨다.



- 포인터는 변수의 이름을 사용할 수 없더라도 주소로 변수에 접근할 수 있는 방법을 제공한다.

# 포인터의 기본\_포인터란?

## ❖ 포인터의 선언(1/2)

- 데이터형과 \*를 쓴 다음 변수 이름을 적어준다.
  - 데이터형
    - 포인터가 가리키는 변수의 데이터형
  - \* : 포인터 수식어

### 형식

데이터형 \*변수명;  
데이터형 \*변수명 = 초기값;

### 사용예

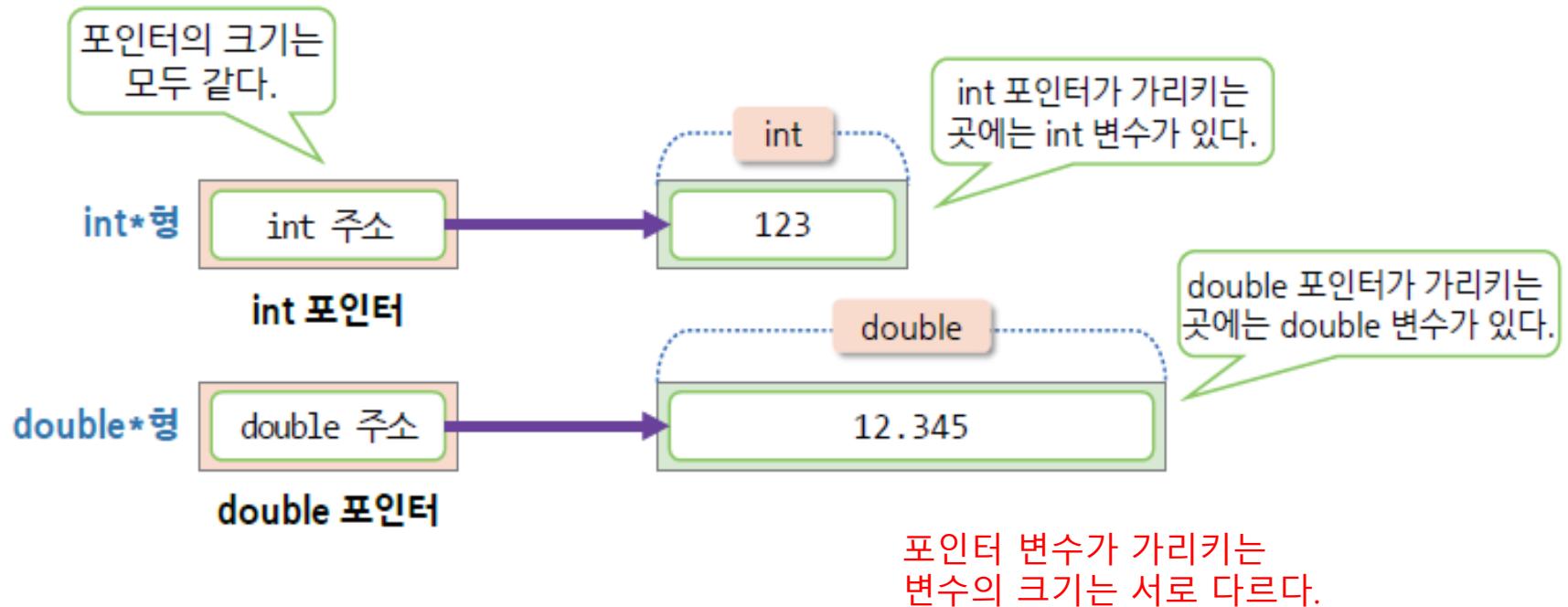
```
int *p;  
double x;  
double *pd = &x;  
char *ptr = NULL;
```

- 어떤 형의 변수를 가리키는지에 따라 포인터의 데이터형이 결정됨
  - int 포인터는 int\*형의 포인터이며, int 변수를 가리킨다.

# 포인터의 기본\_포인터란?

## ❖ 포인터의 선언(2/2)

- 데이터형에 관계없이 포인터의 크기는 항상 같다.
  - 포인터(주소)의 크기는 플랫폼에 의해서 결정된다.
  - 32비트 플랫폼에서 포인터의 크기는 4바이트이다.



# 포인터의 기본\_포인터란?

## ❖ 포인터 변수와 포인터형의 크기

```
01: /* Ex08_01.c */
02: #include <stdio.h>
03:
04: int main(void)
05: {
06:     char *pc;
07:     int *pi;
08:     double *pd;           ← 포인터 변수의 선언
09:
10:    printf("pc의 크기 : %d\n", sizeof(pc));
11:    printf("pi의 크기 : %d\n", sizeof(pi));
12:    printf("pd의 크기 : %d\n", sizeof(pd));   ← 포인터 변수
13:                                ← 크기 구하기
14:    printf("char* 의 크기 : %d\n", sizeof(char*));
15:    printf("short* 의 크기 : %d\n", sizeof(short*));
16:    printf("int* 의 크기 : %d\n", sizeof(int*));
17:    printf("float* 의 크기 : %d\n", sizeof(float*));
18:    printf("double*의 크기 : %d\n", sizeof(double*));   ← 포인터 형의
19:                                ← 크기 구하기
20:    return 0;
21: }
```

### 실행 결과

```
pc 의 크기 : 4
pi 의 크기 : 4
pd 의 크기 : 4
char* 의 크기 : 4
short* 의 크기 : 4
int* 의 크기 : 4
float* 의 크기 : 4
double* 의 크기 : 4
```

# 포인터의 기본\_포인터란?

## ❖ 포인터 변수와 포인터형의 크기

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

```
03 int main(void)
```

```
04 {
```

```
05     int *pi;
```

```
06     double *pd;
```

```
07     char *pc;
```

포인터의 크기는  
데이터형에 관계없이  
항상 같다.

```
09     printf("sizeof(pi) = %d\n", sizeof(pi));
```

```
10     printf("sizeof(pd) = %d\n", sizeof(pd));
```

```
11     printf("sizeof(pc) = %d\n", sizeof(pc));
```

```
13     printf("sizeof(int*) = %d\n", sizeof(int*));
```

```
14     printf("sizeof(double*) = %d\n", sizeof(double*));
```

```
15     printf("sizeof(char*) = %d\n", sizeof(char*));
```

```
16 }
```

32비트 플랫폼의  
실행 결과

실행 결과

sizeof(pi) = 4

sizeof(pd) = 4

sizeof(pc) = 4

sizeof(int\*) = 4

sizeof(double\*) = 4

sizeof(char\*) = 4

포인터의 크기를  
구한다.

포인터형의 크기를  
구한다.

# 포인터의 기본\_포인터란?

## ❖ 포인터의 초기화

- 주소 구하기(**address-of**) 연산자
  - 포인터를 초기화하려면 주소가 필요하다.
  - 변수의 주소를 구하려면 &를 이용한다.
- 널 포인터(**NULL**)
  - 표준 C 라이브러리에 0으로 정의된 매크로 상수
  - 포인터를 어떤 변수의 주소로 초기화할지 알 수 없으면 NULL로 초기화
  - NULL은 메모리 0번지가 아니라 포인터가 어떤 변수도 가리키지 않는다는 뜻이다.

```
int a = 10;  
int *p = &a;
```

p를 a의 주소로  
초기화한다.

```
int *q = NULL;
```

어떤 변수도 가리키지  
않는다.

```
int *r = 0;
```

NULL 대신 0을 사용  
할 수도 있다.

# 포인터의 기본\_포인터란?

## ❖ 포인터의 선언 및 초기화

```
01 #include <stdio.h>
02
03 int main(void)
04 {
05     int a = 10;
06     int *p = &a;           ----- p는 a를 가리킨다.
07     int *q = NULL;        ----- 어떤 변수도 가리키지 않는다
08     int *r = 0;
09
10    printf("p = %p\n", p);
11    printf("q = %p\n", q);
12    printf("r = %p\n", r);
13 }
```

### 실행 결과

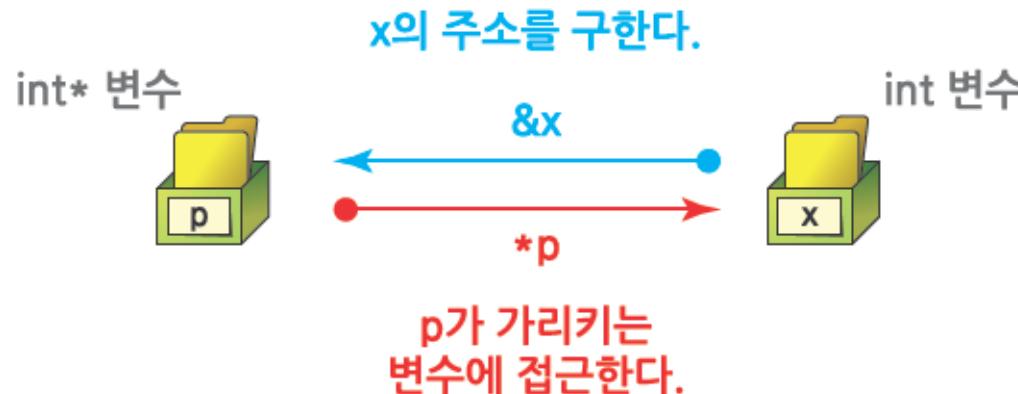
```
p = 00FFF948
q = 00000000
r = 00000000
```

주소를 16진수로  
출력한다.

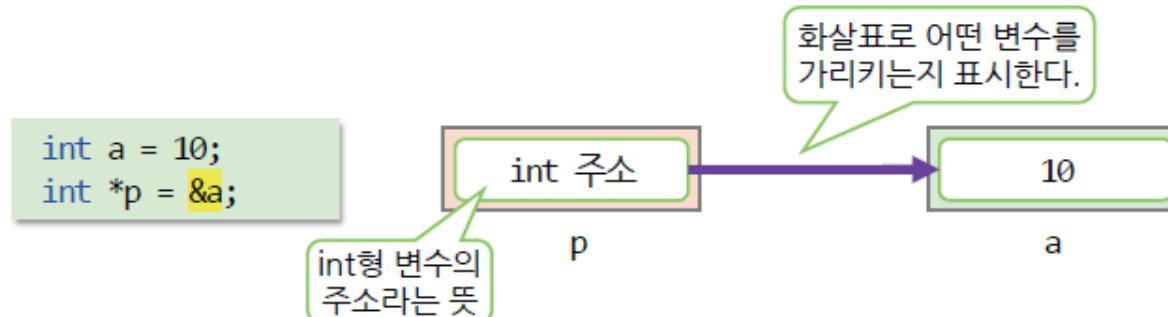
# 포인터의 기본\_포인터란?

## ❖ 포인터의 사용(1/4)

- 포인터를 사용하려면 주소 구하기 연산자(address-of operator)와 간접 참조 연산자(indirection operator)가 필요하다.



- 'type 주소'는 type형 변수의 주소, type 포인터라는 뜻이다.



# 포인터의 기본\_포인터란?

## ❖ 포인터의 사용(2/4)

- 주소 구하기 연산자 & : & 다음에 나오는 변수의 주소를 구하는 데 사용
- & 연산의 결과는 주소이다.
  - & 다음에 있는 변수에 대한 포인터형

```
int a = 10;  
int *p = &a;
```

int\* 형의 주소

- & 연산자는 반드시 변수와 함께 사용해야 하며, 상수나 수식에는 사용할 수 없다.

리터럴 상수의 주소는  
구할 수 없다.



p = &10;

수식의 주소는  
구할 수 없다.



p = &(a + 1);



p = &MAX;

매크로 상수의 주소는  
구할 수 없다.

# 포인터의 기본\_포인터란?

## ❖ 포인터의 사용(3/4)

- **역참조 연산자 \*** : 포인터 변수가 가리키는 변수에 접근해서 변수의 값을 읽어오거나 변경하는 데 사용
  - 포인터가 가리키는 변수를 끌어와서 사용하는 것
  - 간접 참조(indirection) 연산자
  - \* 연산의 결과는 포인터가 가리키는 변수

```
*p = 20;  
printf("*p = %d\n", *p);
```

int형의 변수

int형의 변수

- 역참조 연산자를 이용하면 변수의 이름을 모르더라도 주소로 변수에 접근 가능 : \* 연산자는 반드시 포인터 변수 앞에만 사용할 수 있음

일반 변수에는 \* 연산자를 사용할 수 없다.



```
int a = 10;
```

```
*a = 20;
```



```
*(a + 1) = 20;
```

상수나 수식에는 \* 연산자를 사용할 수 없다.

# 포인터의 기본\_포인터란?

## ❖ 포인터의 사용(3/4)

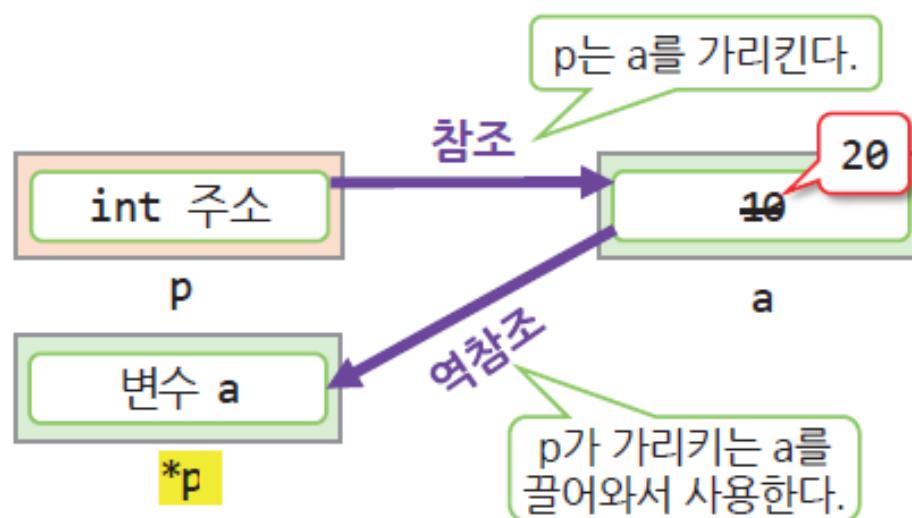
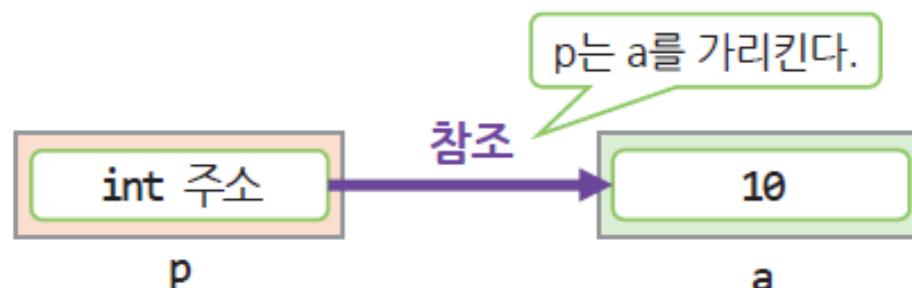
- 역참조 연산자를 사용하면 포인터가 가리키는 변수가 대신 사용됨
  - p가 a를 가리킬 때, \*p 대신 a가 사용되는 것처럼 처리된다.

```
int a = 10;  
int *p = &a;
```

p는 a를 참조한다.

```
*p = 20;  
printf("%d", *p);
```

\*p 대신 a가 사용되는 것처럼 처리된다.



# 포인터의 기본\_포인터란?

## ❖ 포인터 변수의 선언 및 사용 예

```
01: /* Ex08_02.c */
02: #include <stdio.h>
03:
04: int main(void)
05: {
06:     int x; ← int형 변수의 선언
07:     int *p; ← int*형 변수의 선언
08:
09:     p = &x; ← p는 x의 주소 저장
10:     *p = 10; ← p가 가리키는 변수에 접근
11:
12:     printf("*p = %d\n", *p);
13:     printf("x = %d\n", x);
14:
15:     printf("p = %p\n", p);
16:     printf("&x = %p\n", &x);
17:
18:     printf("&p = %p\n", &p);
19:
20:     return 0;
21: }
```

### 실행 결과

```
*p = 10
x = 10
p = 0012FF7C
&x = 0012FF7C
&p = 0012FF78
```

# 포인터의 기본\_포인터란?

## ❖ 포인터 변수의 선언 및 사용 예

```
01 #include <stdio.h>
03 int main(void)
04 {
05     int a = 10;
06     int* p = &a;
08     printf(" a = %d\n", a);
09     printf("&a = %p\n", &a);
11     printf(" p = %p\n", p);
12     printf("*p = %d\n", *p);
13     printf("&p = %p\n", &p);
15     *p = 20;
16     printf("*p = %d\n", *p);
17 }
```

p는 a를 가리킨다.

a의 주소를 출력한다.

p가 가리키는 int형 변수를 출력한다.

포인터에도 주소가 있다.

p가 가리키는 변수에 대입한다.

p가 가리키는 변수를 출력한다.

### 실행 결과

```
a = 10
&a = 004FF71C
p = 004FF71C
*p = 10
&p = 004FF710
*p = 20
```

# 포인터의 사용\_포인터의 용도

## ❖ 다른 함수의 지역변수 변경

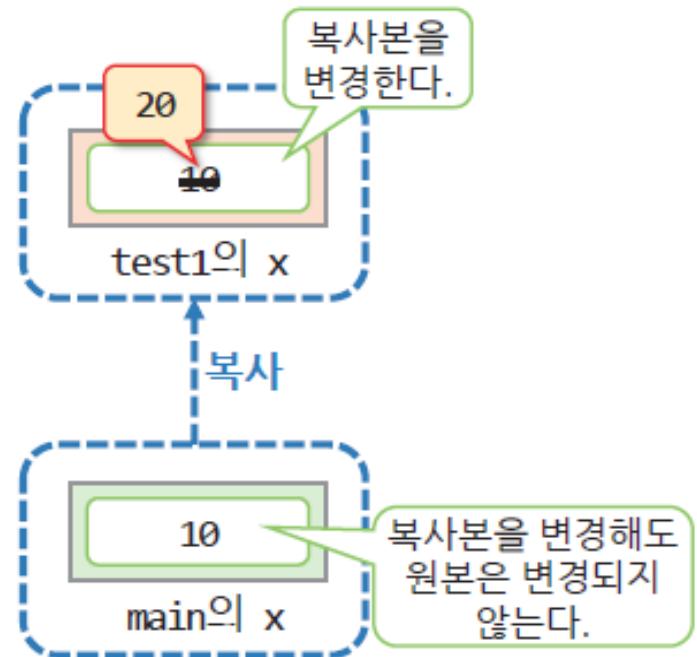
- 인자를 매개변수로 복사해서 전달하는 경우에는 매개변수를 변경해도 원본은 변경되지 않음

```
void test1(int x)
{
    x = 20;           복사본을 변경한다.

}               test1이 리턴할 때
                test1의 x는 소멸된다.

int main(void)
{
    int x = 10;      main의 x를 복사
                    해서 전달한다.
    test1(x);
    printf("x = %d\n", x);
}

원본은 변경
되지 않는다.
```



# 포인터의 사용\_포인터의 용도

## ❖ 다른 함수의 지역변수 변경

- 변수를 직접 사용할 수 없을 때 포인터를 이용해서 주소로 접근

```
void test2(int *p)
{
    *p = 20;
}

int main(void)
{
    int x = 10;
    test2(&x);
    printf("x = %d\n", x);
}
```

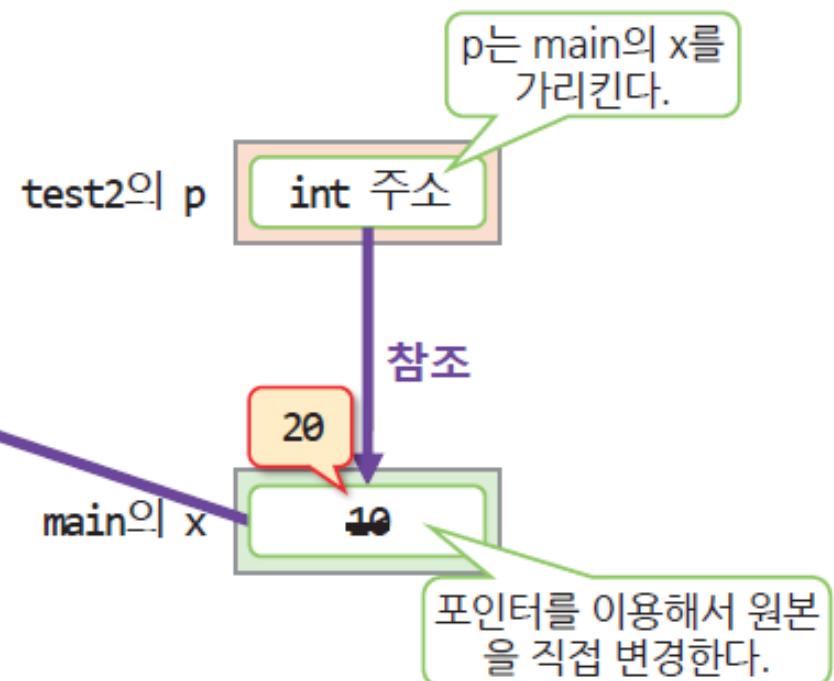
x의 주소로 초기화된다.

p가 가리키는 main의 x를 변경한다.

역참조

x의 주소를 전달한다.

원본이 변경된다.



# 포인터의 사용\_포인터의 용도

## ❖ 포인터가 필요한 경우(1/2)

```
01 #include <stdio.h>
02
03 void test1(int x)
04 {
05     x = 20;
06 }
07
08 void test2(int *p)
09 {
10     *p = 20;
11 }
12
13 int main(void)
14 {
```

매개변수 x는 함수가 호출될 때  
main의 x로 초기화된다.

매개변수 x를 변경해도 함수가  
리턴될 때 x가 소멸된다.

매개변수 p는 main의 x를  
가리킨다.

p가 가리키는 변수(main의 x)  
를 변경한다.

# 포인터의 사용\_포인터의 용도

## ❖ 포인터가 필요한 경우(2/2)

```
15 int x = 10;
16 test1(x);
17 printf("test1 호출 후 x = %d\n", x); x의 값은 변경되지 않는다.
18
19 test2(&x);
20 printf("test2 호출 후 x = %d\n", x); x의 값이 변경된다.
21 }
```

x의 주소를 test2의 매개변수 p로 전달한다.

main의 x를 test1의 매개변수 x로 복사해서 전달한다.

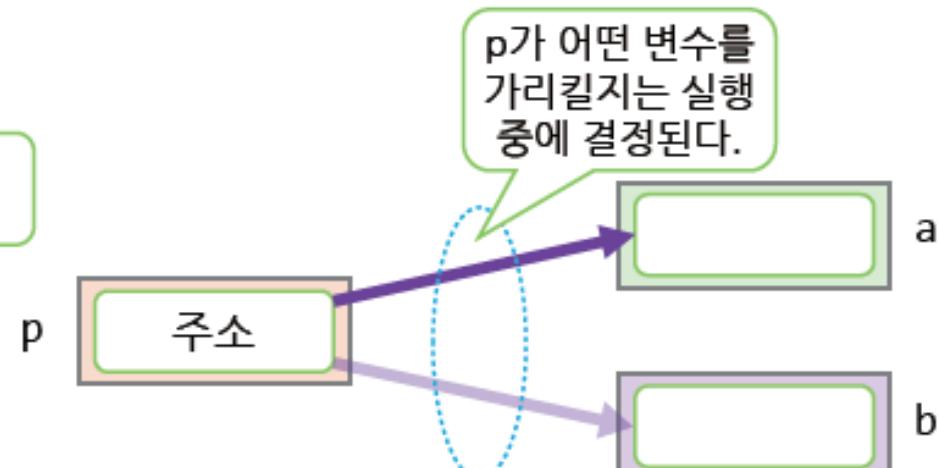
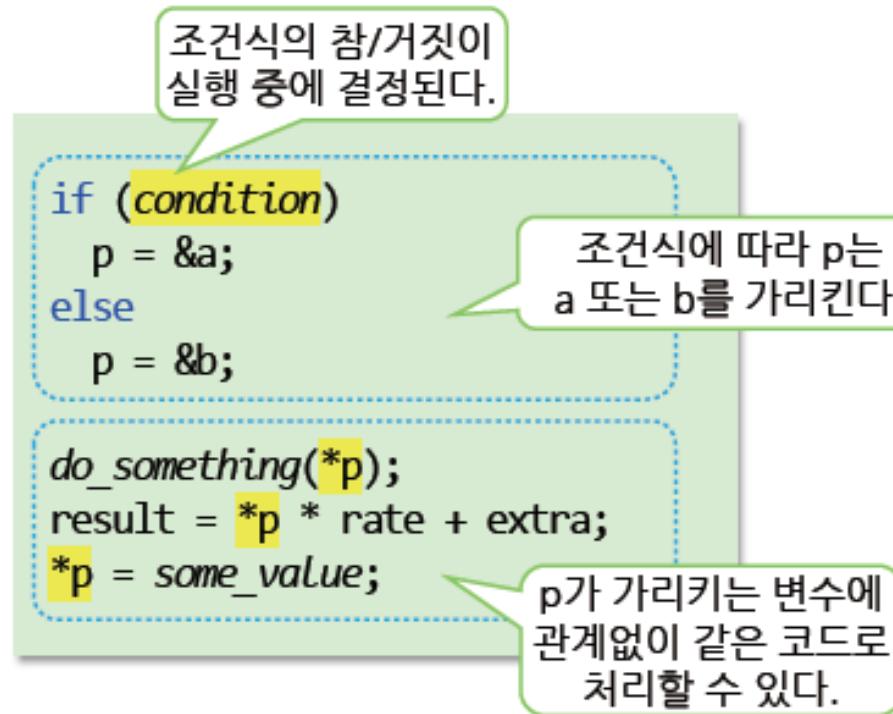
### 실행 결과

```
test1 호출 후 x = 10
test2 호출 후 x = 20
```

# 포인터의 사용\_포인터의 용도

❖ 포인터를 이용하면 여러 변수에 대하여 공통의 코드를 작성 할 수 있음

- 포인터가 어떤 변수를 가리킬지 미리 알 수 없는 경우에도 포인터가 가리키는 변수로 어떤 작업을 수행하도록 코드를 작성할 수 있음



# 포인터의 사용\_포인터 사용시 주의 사항

## ❖ 잘못된 포인터(1/2)

- 포인터도 변수이므로 반드시 초기화해야 한다.
- 포인터 변수를 초기화하지 않고 사용하면 실행 에러가 발생한다.

`int *p;` ○ p는 초기화되지 않았으므로 쓰레기 값을 가진다.

`*p = 10;` ○ 실행 에러가 발생한다.

- **널 포인터** : 포인터가 다른 변수를 가리키지 않을 때는 NULL(0)로 초기화한다.

`int *p = NULL;` ○ p를 널 포인터로 초기화한다.

p는 아직 다른 변수를 가리키지 않는다.

`int* q = NULL;`



`printf("%d", *q);`

널 포인터로 역참조 연산을 수행하면 프로그램이 죽는다.

# 포인터의 사용\_포인터 사용시 주의 사항

## ❖ 잘못된 포인터(2/2)

- 포인터를 안전하게 사용하려면 우선 포인터가 널 포인터인지를 검사한다.

q가 널 포인터가 아닌 경우에  
만 역참조 연산한다.

```
if(q != NULL)  
    printf("%d", *q);
```

```
if( p )  
    *p = 10;
```

p가 널 포인터가 아닌지 확인한다.

# 포인터의 사용\_포인터 사용시 주의 사항

## ❖ 포인터형과 변수의 데이터형

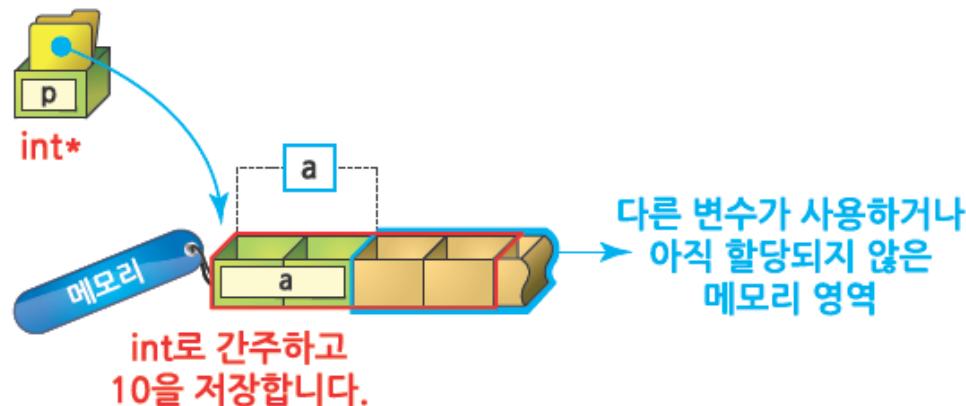
- 포인터 변수의 데이터형이 반드시 포인터 변수가 가리키는 변수의 데이터형과 일치해야 한다.

```
short a;
```

```
int *p = &a; ○———— 컴파일 경고가 발생한다.
```

```
*p = 10; ○———— 컴파일 경고를 무시하고 실행하면, 실행 에러가 발생한다.
```

```
short a;  
int *p = &a;  
  
*p = 10;
```

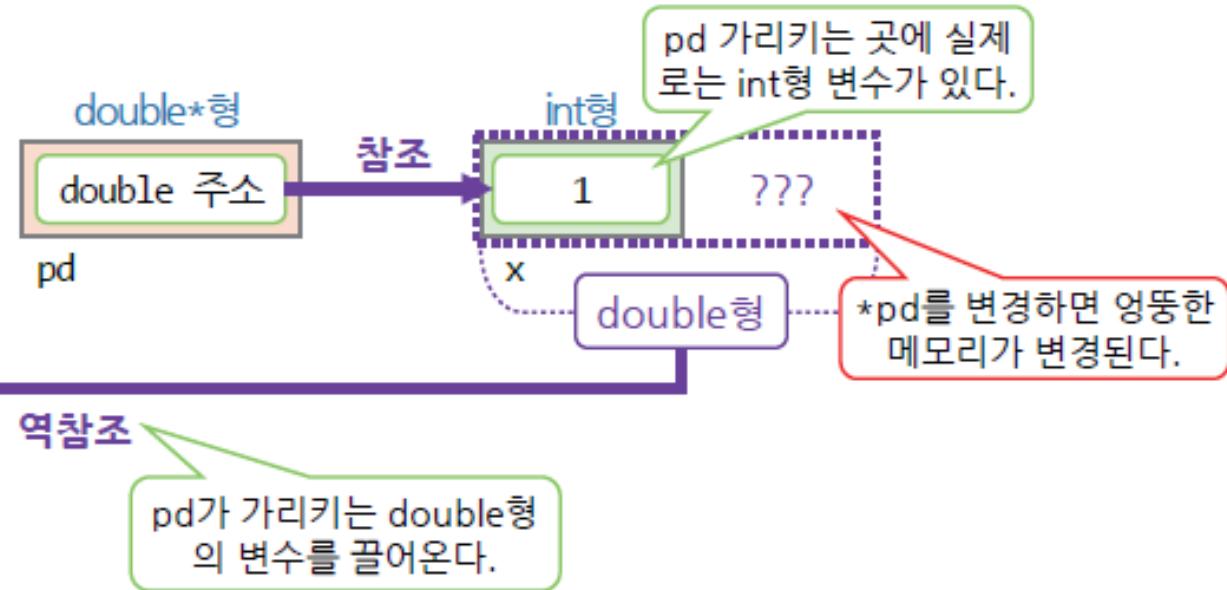


# 포인터의 사용\_포인터 사용시 주의 사항

## ❖ 포인터형과 변수의 데이터형

- 포인터형이 일치하지 않으면 컴파일 경고가 발생하며, 컴파일 경고를 무시하고 실행하면 실행 에러 발생

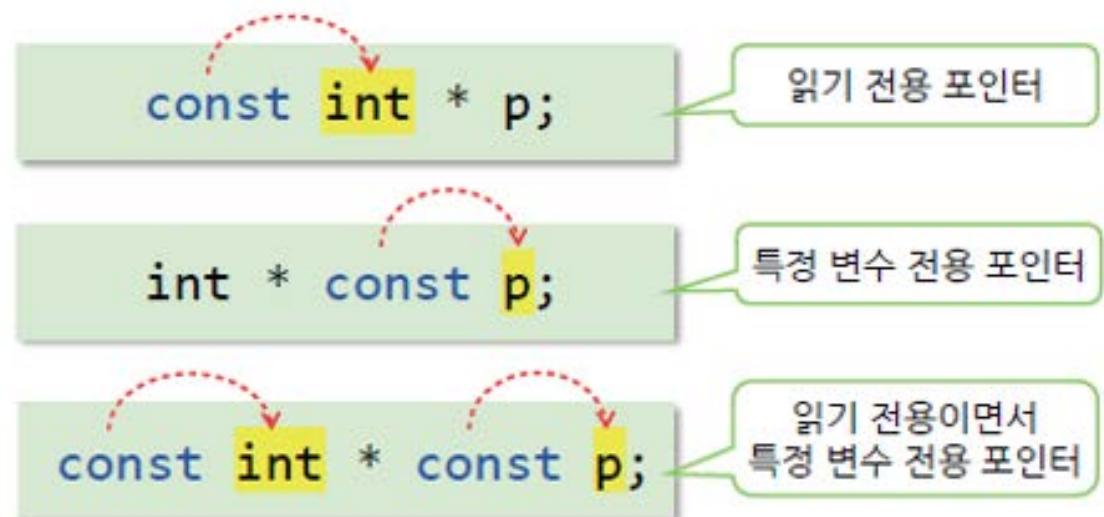
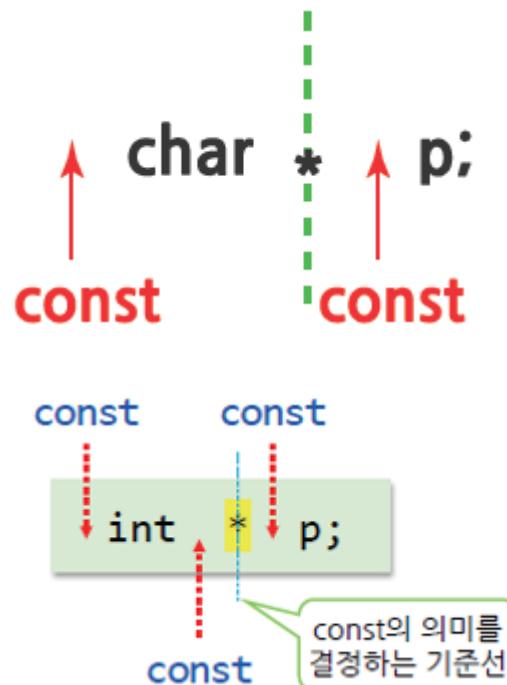
```
int x = 1;  
double *pd = NULL;  
  
pd = &x;      // 컴파일 경고  
*pd = 12.34; // 실행 에러
```



# 포인터\_const 포인터

## ❖ const 포인터

- 포인터 변수를 선언할 때도 **const** 키워드를 사용할 수 있다.
- const**가 사용되는 위치에 따라 포인터 변수의 의미가 달라진다.
  - 세 가지 **const** 포인터 중에서 읽기 전용 포인터만 주로 사용됨



# 포인터\_const 포인터

## ❖ const 키워드가 데이터형 앞에 있는 경우

- 포인터가 가리키는 변수의 값을 읽어볼 수만 있고 변경할 수 없다.
- 포인터가 가리키는 변수를 **const** 변수인 것처럼 사용

```
int a = 10, b = 20;  
const int *p1 = &a;
```

p1은 a에 읽기 전용으로 접근한다.

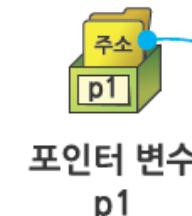
```
printf("*p1 = %d\n", *p1);
```

p1이 가리키는 변수의 값을 읽어온다.

```
*p1 = 100;
```

p1이 가리키는 변수의 값을 변경할 수는 없다.

```
const char *p1 = str1;
```



- 포인터 변수 자신의 값(주소)은 변경할 수 있다.

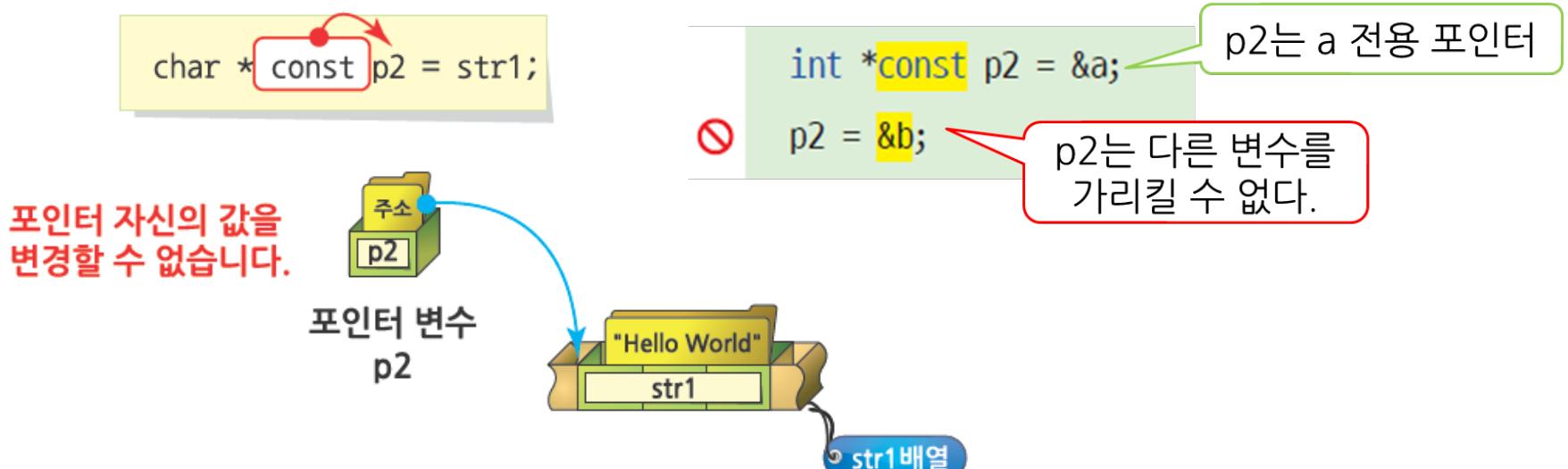
```
p1 = &b;
```

p1이 다른 변수를 가리킬 수 있다.

# 포인터\_const 포인터

## ❖ const 키워드가 포인터 변수명 앞에 있는 경우

- 포인터 변수 자신의 값(주소)을 변경할 수 없다.
  - 특정 변수의 전용 포인터
  - 선언 시 특정 변수의 주소로 초기화해야 하며, 초기화 후에 다른 변수를 가리킬 수 없다.



- 포인터가 가리키는 변수의 값은 변경할 수 있다.

\*p2 = 100;

p2가 가리키는 변수의 값을 변경할 수 있다.

# 포인터\_const 포인터

## ❖ const 키워드가 양쪽 모두에 있는 경우

- 읽기 전용 포인터이면서 특정변수의 전용 포인터
- 반드시 초기화해야 한다.
- 포인터가 가리키는 변수의 값도 변경할 수 없고 포인터 변수 자신의 값(주소)도 변경할 수 없다.

```
const int * const p3 = &a;
```

🚫 \*p3 = 100;

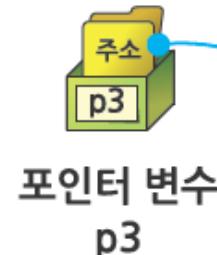
p3 = &b;

p3가 가리키는 변수의 값을  
변경할 수는 없다.

p3가 다른 변수를  
가리킬 수 없다.

```
const char * const p3 = str1;
```

포인터 자신의 값을  
변경할 수 없습니다.



포인터가 가리키는 값을  
변경할 수 없습니다.



# 포인터\_const 포인터

```
01 #include <stdio.h>
02
03 int main(void)
04 {
05     int a = 10, b = 20;
06
07     const int *p1 = &a;
08     int *const p2 = &a;
09     const int * const p3 = &a;
10
11     printf("*p1 = %d\n", *p1);
12     /*p1 = 100;
13     p1 = &b;
14     printf("*p1 = %d\n", *p1);
```

p1은 a에 읽기 전용으로 접근한다.

p2는 a 전용 포인터이다.

p3는 읽기 전용이면서 a 전용 포인터

p1이 가리키는 변수의 값을 변경할 수는 없다.

# 포인터\_const 포인터

```
15  
16     //p2 = &b;  
17     *p2 = 100;  
18     printf("*p2 = %d\n", *p2);  
19  
20     /*p3 = 100;  
21     //p3 = &b;  
22     printf("*p3 = %d\n", *p3);  
23 }
```

p2는 다른 변수를  
가리킬 수 없다.

p3가 가리키는 변수의 값을  
변경할 수 없고, p3가 다른  
변수를 가리킬 수 없다.

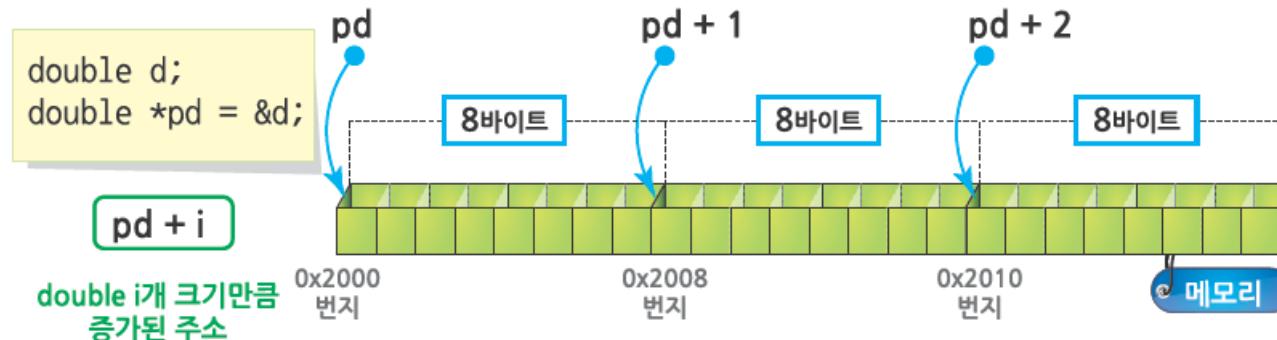
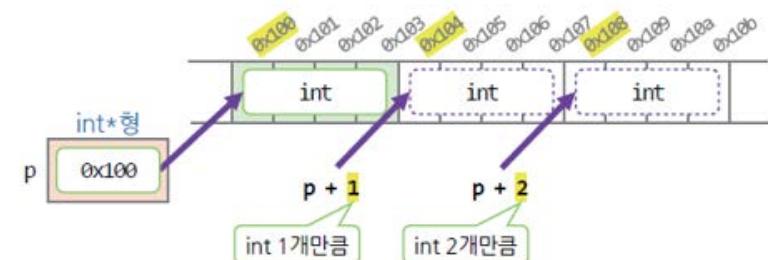
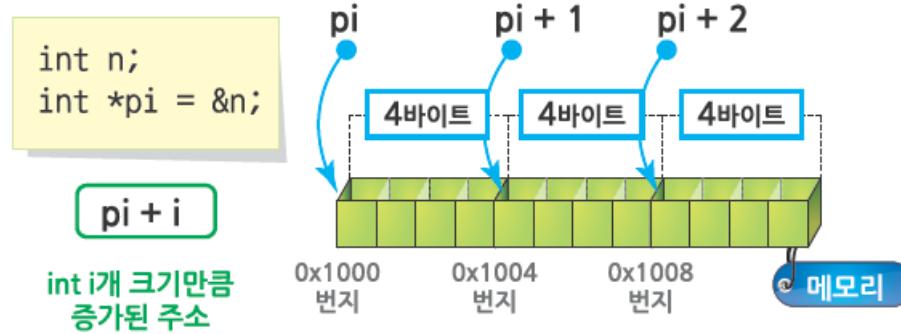
## 실행 결과

```
*p1 = 10  
*p1 = 20  
*p2 = 100  
*p3 = 100
```

# 배열과 포인터의 관계\_포인터의 연산

## ❖ 포인터와 +, - 연산(1/2)

- p+N은 p가 가리키는 데이터형 N개 크기만큼 증가된 주소가 연산의 결과이다.
- p-N은 p가 가리키는 데이터형 N개 크기만큼 감소된 주소가 연산의 결과이다.



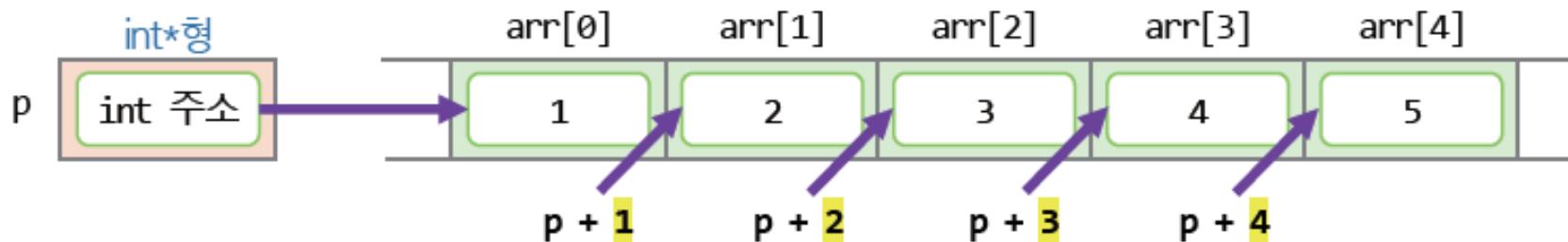
# 배열과 포인터의 관계\_포인터의 연산

## ❖ 포인터와 +, - 연산(2/3)

- ‘포인터+정수’ 연산은 포인터가 가리키는 주소에 마치 배열이 있는 것처럼 메모리에 접근
- 포인터 p가 배열 arr를 가리킬 때  $*(p+i)$ 는  $arr[i]$ 를 의미

```
int arr[5] = { 1, 2, 3, 4, 5 };
int *p = &arr[0];
```

p는 arr[0]을  
가리킨다.



$$p + i == \&arr[i]$$

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

# 배열과 포인터의 관계\_포인터의 연산

## ❖ 포인터와 +, - 연산(3/3)

- ‘포인터-포인터’ 연산은 두 포인터의 차를 구하는 데 사용된다.
  - 포인터가 가리키는 데이터형 몇 개 크기만큼 차이가 나는지 구함

```
int arr[5];
```

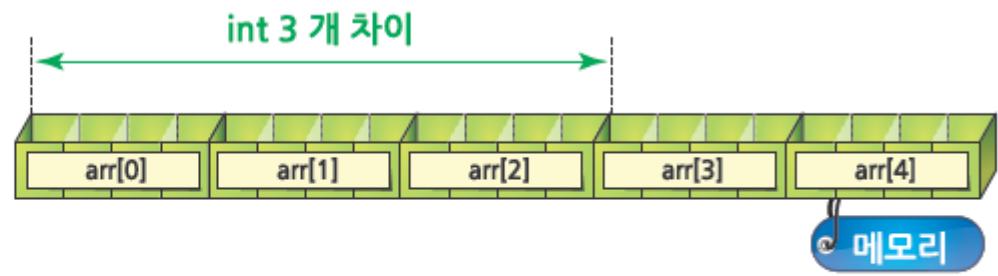
```
int diff = &arr[3] - &arr[0];
```

arr[3]과 arr[0]은 int 3개만큼 떨어져 있다.

```
int arr[5];
int diff = &arr[3] - &arr[0];
```

**&arr[3] - &arr[0]**

int 3개만큼 주소가 차이나므로  
연산의 결과는 3이 됩니다.



# 배열과 포인터의 관계\_포인터의 연산

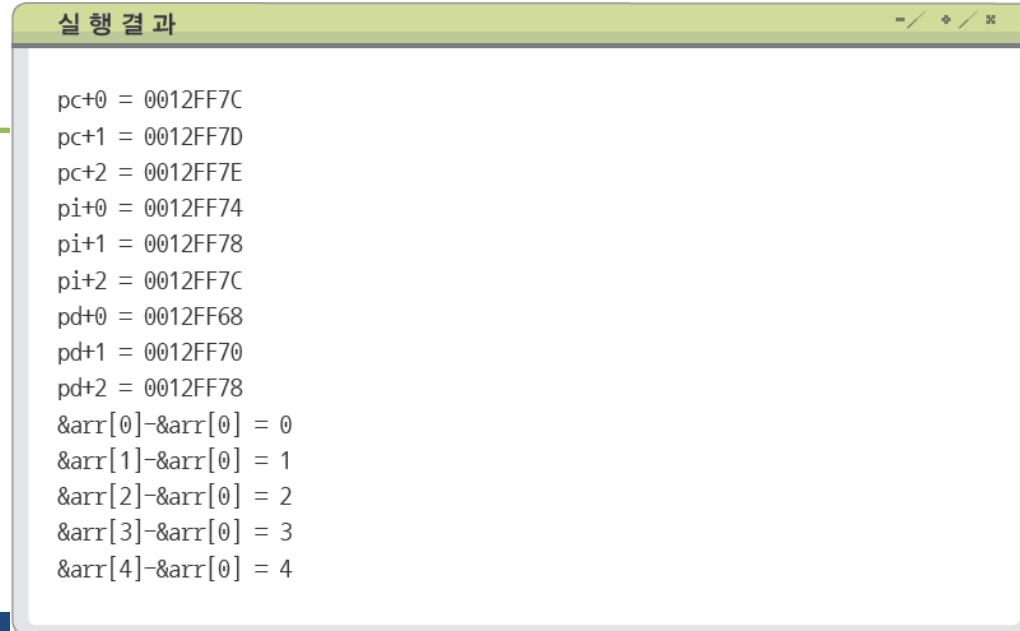
## ❖ 포인터의 더하기, 빼기 예(1/2)

```
01: /* Ex08_03.c */
02: #include <stdio.h>
03:
04: int main(void)
05: {
06:     char ch;
07:     char *pc = &ch; ← char*형 변수의 선언
08:
09:     int n;
10:     int *pi = &n; ← int*형 변수의 선언
11:
12:     double d;
13:     double *pd = &d; ← double*형 변수의 선언
14:
15:     int arr[3];
16:     int i;
17:
18:     for(i = 0 ; i < 3 ; i++)
19:         printf("pc+%d = %p\n", i, pc+i); ← char*형에 정수를 더하는 연산
20:
```

# 배열과 포인터의 관계\_포인터의 연산

## ❖ 포인터의 더하기, 빼기 예(2/2)

```
21:     for(i = 0 ; i < 3 ; i++)
22:         printf("pi+%d = %p\n", i, pi+i); ← int*형에 정수를 더하는 연산
23:
24:     for(i = 0 ; i < 3 ; i++)
25:         printf("pd+%d = %p\n", i, pd+i); ← double*형에 정수를 더하는 연산
26:
27:     for(i = 0 ; i < 5 ; i++)
28:         printf("&arr[%d]-&arr[0] = %d\n", i, &arr[i]-&arr[0]); ← 포인터의 차를
29:                                         구하는 연산
30:     return 0;
31: }
```



# 배열과 포인터의 관계\_포인터의 연산

## ❖ '포인터+정수'연산의 결과

```
01 #include <stdio.h>
02
03 int main(void)
04 {
05     int *p = (int*)0x100;
06     double *q = (double*)0x100;
07     char *r = (char*)0x100;
08
09     printf("int* : %p, %p, %p\n", p, p + 1, p + 2);
10    printf("double*: %p, %p, %p\n", q, q + 1, q + 2);
11    printf("char* : %p, %p, %p\n", r, r + 1, r + 2);
12 }
```

### 실행 결과

```
int* : 00000100, 00000104, 00000108
double*: 00000100, 00000108, 00000110
char* : 00000100, 00000101, 00000102
```

포인터 연산 결과를 확인하기  
위해 절대 주소로 초기화

# 배열과 포인터의 관계\_포인터의 연산

## ❖ 포인터와 ++, -- 연산(1/2)

- 포인터에 대한 증감 연산(++, --)도 포인터의 데이터형에 의해 연산의 결과가 결정된다.

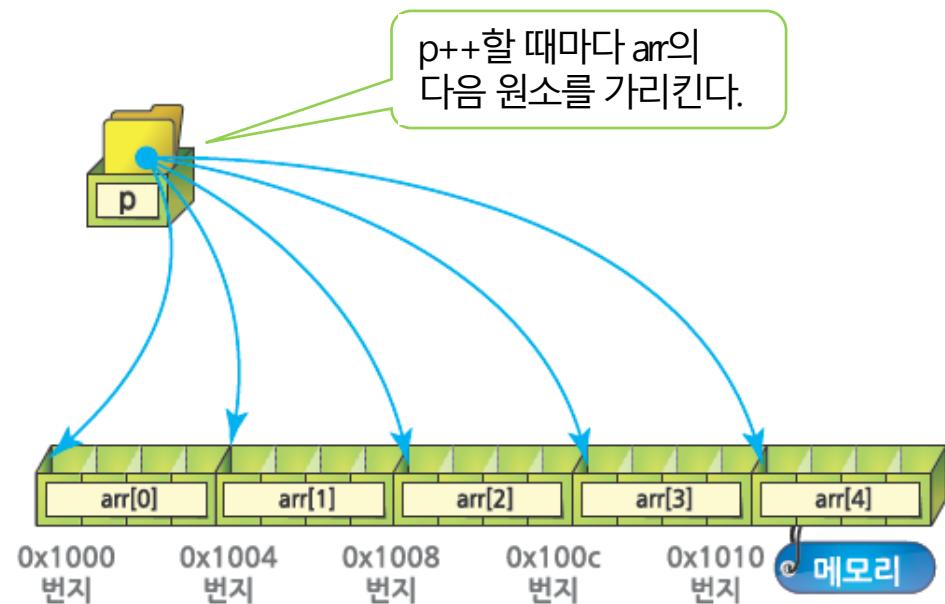
```
int arr[5] = { 1, 2, 3, 4, 5 };
int *p = &arr[0];
for (i = 0; i < 5; i++, p++)
{
    printf("p= %p, ", p);
    printf("*p = %d\n", *p);
}
```

p가 가리키는 원소를 출력한다.

p++;

sizeof(int)만큼 주소 증가

p++할 때마다 arr의 다음 원소를 가리킨다.



i = 0일 때, p는 0x1000번지이고, \*p는 arr[0]입니다.  
i = 1일 때, p는 0x1004번지이고, \*p는 arr[1]입니다.  
i = 2일 때, p는 0x1008번지이고, \*p는 arr[2]입니다.  
i = 3일 때, p는 0x100c번지이고, \*p는 arr[3]입니다.  
i = 4일 때, p는 0x1010번지이고, \*p는 arr[4]입니다.

# 배열과 포인터의 관계\_포인터의 연산

## ❖ 포인터의 증감 연산자 이용 예

```
01 #include <stdio.h>
02
03 int main(void)
04 {
05     int arr[5] = { 1, 2, 3, 4, 5 };
06     int *p = &arr[0];
07     int i;
08
09     for (i = 0; i < 5; i++, p++)
10     {
11         printf("p= %p, ", p);
12         printf("*p = %d\n", *p);
13     }
14 }
```

p는 arr[0]을 가리킨다.

for (i = 0; i < 5; i++, p++)

p가 가리키는 원소가 계속 바뀐다.

### 실행 결과

```
p= 0102FBBD8, *p = 1
p= 0102FBDC, *p = 2
p= 0102FBE0, *p = 3
p= 0102FBE4, *p = 4
p= 0102FBE8, *p = 5
```

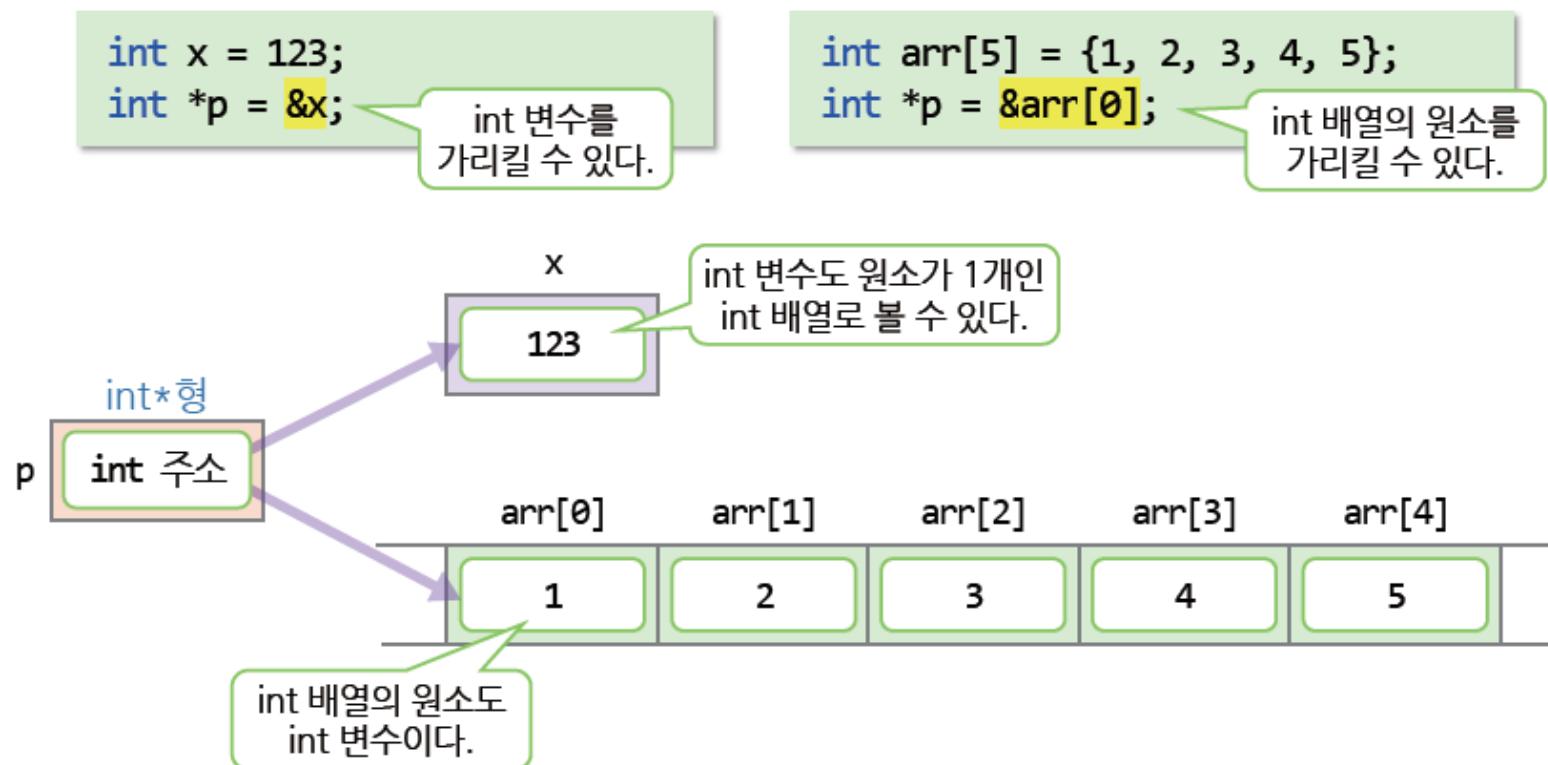
for가 수행되는 동안 p는 arr[0]~arr[4]를 가리킨다.

p가 가리키는 원소가 계속 바뀐다.

# 배열과 포인터의 관계\_배열의 원소를 가리키는 포인터

## ❖ 배열처럼 사용되는 포인터(1/3)

- *type\**형의 포인터는 *type*형의 변수 또는 *type*형 배열의 원소를 가리킬 수 있다.



# 배열과 포인터의 관계\_ 배열의 원소를 가리키는 포인터

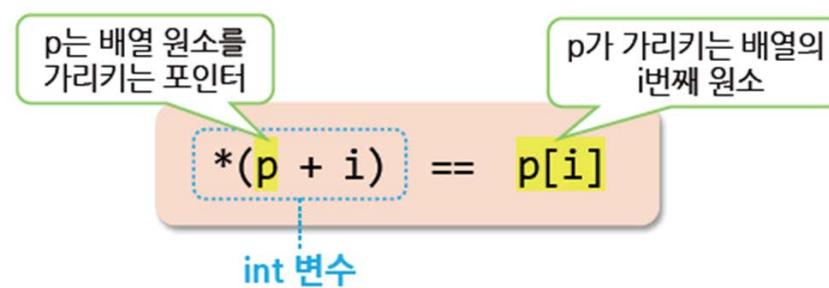
## ❖ 배열처럼 사용되는 포인터(2/3)

- 인덱스 없이 배열명만 사용하면 배열의 시작 주소를 의미한다.

```
int* p = arr;
```

p에 arr 배열의 시작 주소 (&arr[0])를 저장한다.

- 포인터 변수를 배열 이름인 것처럼 사용할 수 있다.
  - $p[i]$ 는 항상  $*(p+i)$ 으로 처리된다.



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

포인터가 가리키는  
곳에서 i개의 원소만큼  
떨어진 주소에 있는 값

포인터가 가리키는  
배열의 i번째 원소

$$p + i = \&p[i]$$

포인터가 가리키는  
곳에서 i개의 원소만큼  
떨어진 주소

포인터가 가리키는 배열의  
i번째 원소의 주소

# 배열과 포인터의 관계\_배열의 원소를 가리키는 포인터

## ❖ 배열처럼 사용되는 포인터(3/3)

- 배열의 시작 주소로 초기화된 포인터를 이용해서 배열의 모든 원소에 접근할 수 있다.
- 배열의 원소를 가리키는 포인터는 배열의 어떤 원소든지 가리킬 수 있다.

```
int arr[5] = {10, 20, 30, 40, 50};  
int *p = &arr[2]; p는 arr[2]를 가리킨다.  
printf("p[0] = %d", p[0]); p[0]은 arr[2]를 의미하므로 30을 출력한다.  
printf("p[1] = %d", p[1]); p[1]은 arr[3]을 의미하므로 40을 출력한다.
```

# 배열과 포인터의 관계\_배열의 원소를 가리키는 포인터

## ❖ 포인터를 배열처럼 사용하는 경우

```
01 #include <stdio.h>
02
03 int main(void)
04 {
05     int arr[5] = { 1, 2, 3, 4, 5 };
06     int *p = arr;           // 배열 이름은
07     int i;                 // 배열의 시작 주소
08
09     for (i = 0; i < 5; i++)
10         printf("p[%d] = %d\n", i, p[i]);
11 }
```

### 실행 결과

```
p[0] = 1
p[1] = 2
p[2] = 3
p[3] = 4
p[4] = 5
```

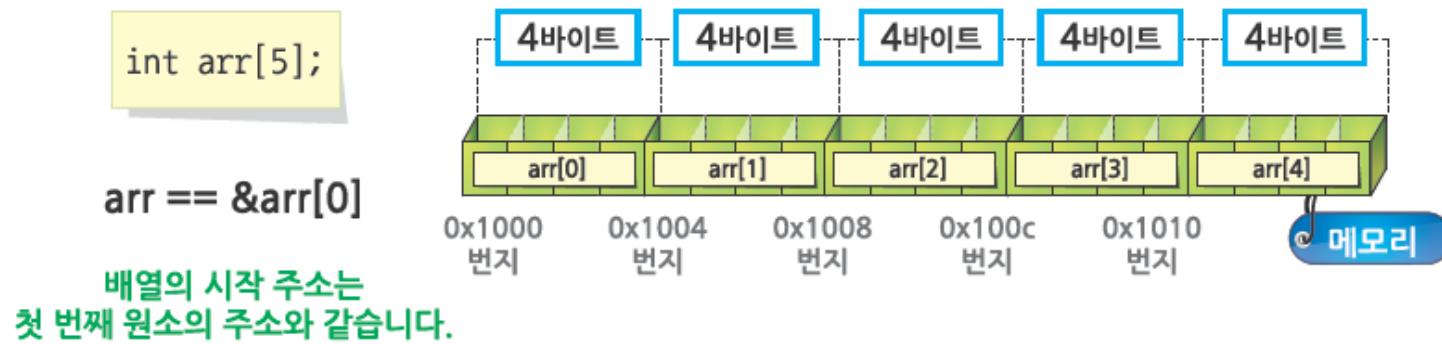
배열 이름은  
배열의 시작 주소

p를 배열처럼  
사용한다.

# 배열과 포인터의 관계\_포인터로서의 배열

## ❖ 포인터처럼 사용되는 배열(1/2)

- 배열 이름은 배열의 시작 주소이므로 배열 이름을 포인터처럼 사용할 수 있다.
- 배열의 시작 주소를 구할 때는 & 없이 배열명만 사용한다.

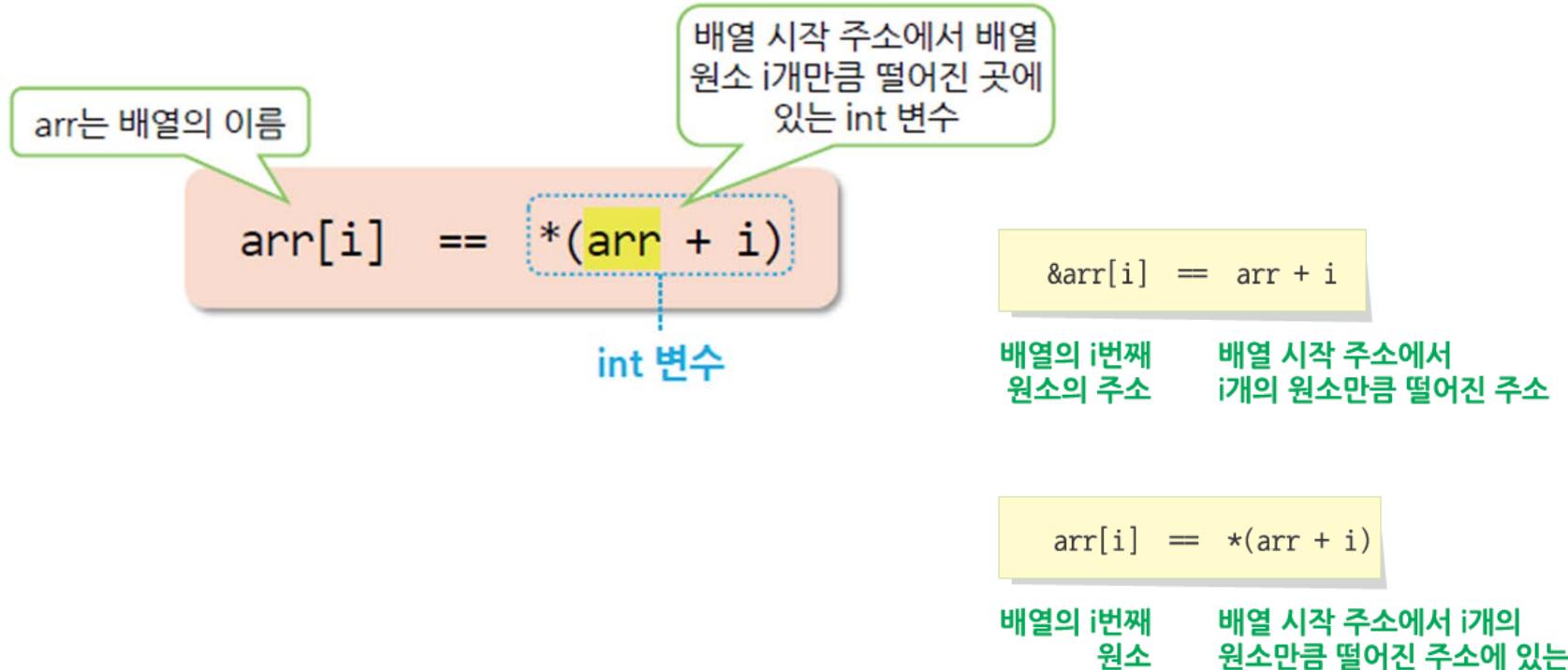


- 배열명은 포인터인 것처럼 사용할 수 있다.
  - 인덱스를 사용하는 대신 배열의 시작 주소로 포인터 연산을 하면 배열의 특정 원소에 접근할 수 있다.

# 배열과 포인터의 관계\_포인터로서의 배열

## ❖ 포인터처럼 사용되는 배열(2/2)

- **\*(arr+i)**는 arr[i]를 의미한다.
  - 배열의 시작 주소에서 int i개 크기만큼 증가된 주소에 있는 값



# 배열과 포인터의 관계\_배열과 포인터의 비교

## ❖ 배열과 포인터의 차이점

- 배열 이름은 특정 변수 전용 포인터로 볼 수 있다.
  - 배열 이름(배열의 시작 주소)에 다른 주소를 대입하거나 배열 이름으로 증감 연산을 할 수 없다.
    - 배열이 메모리에 할당되고 나면, 배열의 시작 주소를 변경할 수 없다.
  - 포인터는 값을 변경할 수 있다.
    - 포인터 변수는 값을 변경할 수 있으므로, 포인터 변수에 보관된 주소는 변경할 수 있다.

x는 배열의 이름

```
int x[5] = { 1, 2, 3, 4, 5 };
int y[5] = { 0 };

x = y; // 컴파일 에러
x++; // 컴파일 에러
```

배열의 시작 주소는 변경할 수 없다.

```
int x[5] = { 1, 2, 3, 4, 5 };
int y[5] = { 0 };
int *p = x;

p = y; // OK
p++; // OK
```

p는 포인터

포인터에 저장된 주소는 다른 주소로 변경할 수 있다.

# 배열과 포인터의 관계\_배열과 포인터의 비교

## ❖ 배열과 포인터의 차이점

- `sizeof(배열명)`은 배열 전체의 바이트 크기를 구하지만, `sizeof(포인터명)`은 포인터 변수의 크기를 구한다.

```
int x[5];
int *p = x;
printf("x의 크기 = %d\n", sizeof(x));
printf("p의 크기 = %d\n", sizeof(p));
```

x 배열 전체의 크기  
이므로 20바이트

포인터 p의 크기이  
므로 4바이트

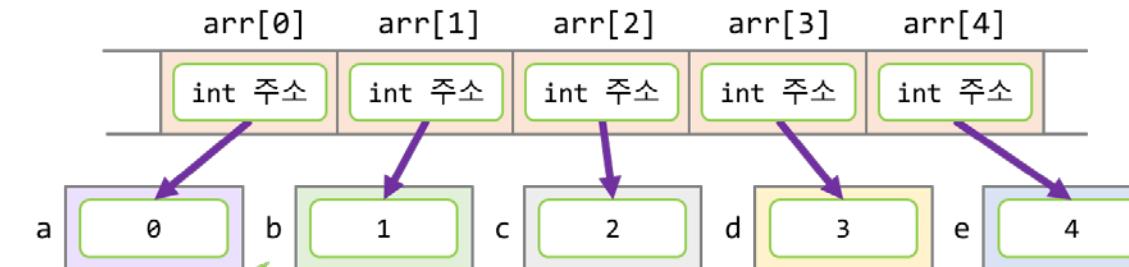
# 여러가지 포인터의 선언

## ❖ 포인터 배열 : 주소를 저장하는 배열



- 포인터 배열의 각 원소가 다른 변수를 가리키는 포인터

```
int a, b, c, d, e;  
int *arr[5] = { &a, &b, &c, &d, &e };  
  
for (i = 0; i < 5; i++)  
    *arr[i] = i;
```



a, b, c, d, e는 서로  
다른 변수이므로 주소  
가 연속되지 않는다.

# 여러가지 포인터의 선언

## ❖ 포인터 배열의 선언 및 사용

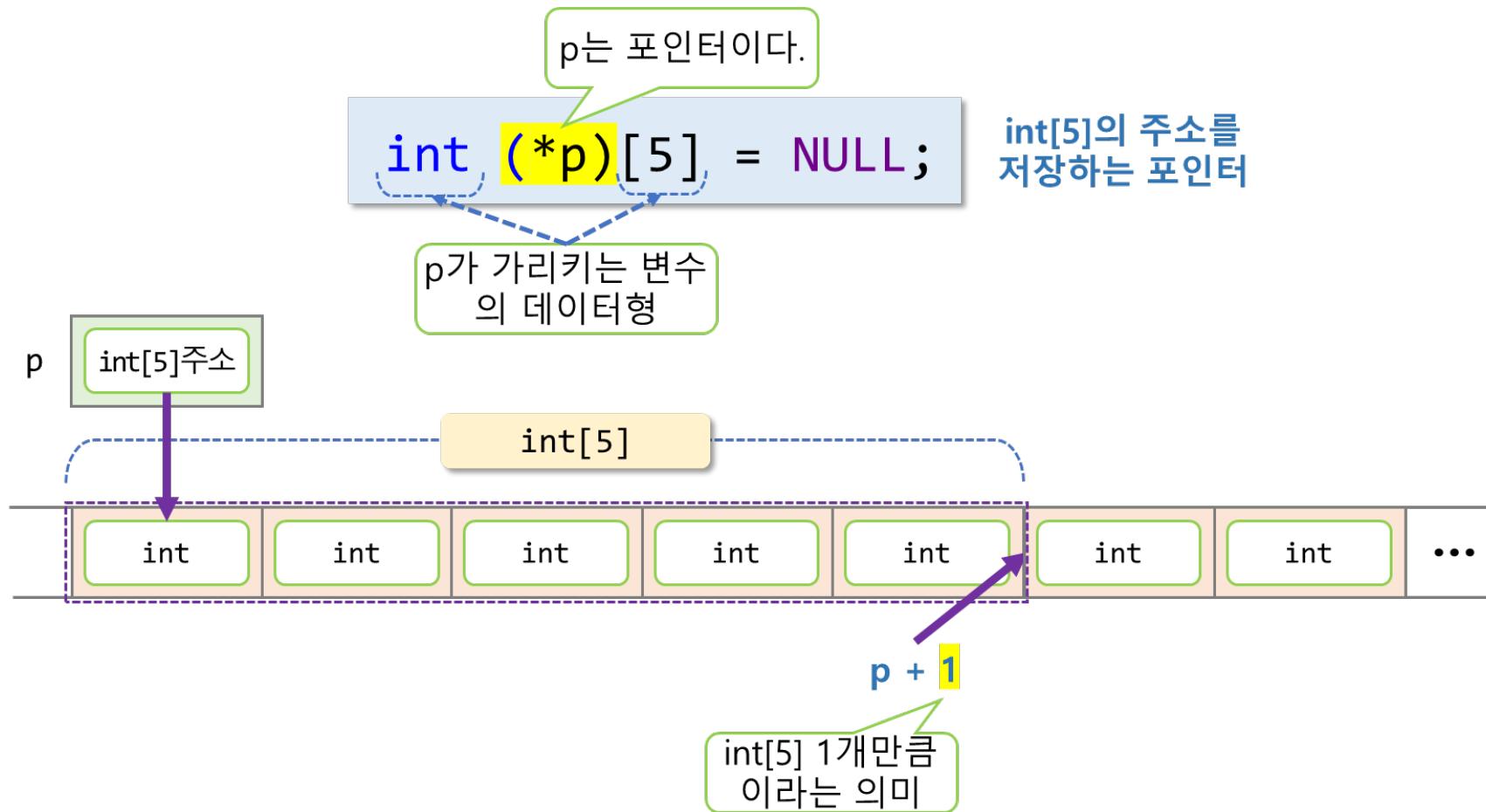
```
03 int main(void)
04 {
05     int a, b, c, d, e;
06     int *arr[5] = { &a, &b, &c, &d, &e };           // 포인터 배열
07     int i;
08
09     for (i = 0; i < 5; i++)
10    {
11        *arr[i] = i;
12        printf("%d ", *arr[i]);                  // arr[i]는 포인터이다.
13    }
14    printf("\n");
15
16    return 0;
17 }
```

실행결과

0 1 2 3 4

# 여러가지 포인터의 선언

## ❖ 배열에 대한 포인터 : 배열 전체를 가리키는 포인터



# 여러가지 포인터의 선언

- ❖ 배열에 대한 포인터는 2차원 배열과 함께 사용된다.
  - 열 크기만큼 만들어진 블록을 가리킬 때 배열에 대한 포인터를 사용
  - 배열에 대한 포인터를 사용할 때는 2차원 배열인 것처럼 사용

배열 원소에 대한 포인터

```
int *p;  
p + 1;
```

배열 원소 크기  
(int)

배열 전체에 대한 포인터

```
int (*p)[5];  
p + 1;
```

배열 전체 크기  
(int [5])

# 여러가지 포인터의 선언

## ❖ 배열에 대한 포인터의 선언 및 사용

```
03 int main(void)
04 {
05     int data[3][5] = {
06         {1, 2, 3, 4, 5},
07         {6, 7, 8, 9, 10},
08         {11, 12, 13, 14, 15}
09     };
10     int(*p)[5] = &data[0];      // int[5] 배열에 대한 포인터
11     int i, j;
12
13     for (i = 0; i < 3; i++)
14     {
15         for (j = 0; j < 5; j++)
16             printf("%2d ", p[i][j]);    // 2차원 배열인 것처럼 사용한다.
17         printf("\n");
18     }
19
20     return 0;
21 }
```

실행결과

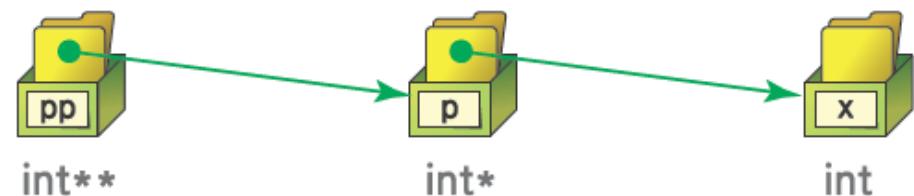
```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

# 여러가지 포인터의 선언

## ❖ 이중 포인터

- 포인터 변수의 주소를 저장하는 포인터 변수

```
int x = 10;  
int *p = &x;      // 포인터  
int **pp = &p;    // 이중 포인터
```



pp가 가리키는  
변수의 데이터형

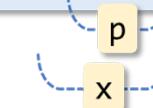
pp는  
포인터 변수

int \*\*pp = &p;

int\*형 변수의 주소를  
저장하는 포인터

- 이중 포인터가 가리키는 포인터를 이용해서 변수에 접근하려면 \*\*처럼 두 번 간접 참조를 해야 한다.

\* \*pp = 20;



# 함수와 포인터\_함수의 인자 전달 방법

구분	특징
값에 의한 호출 (값 호출)	<ul style="list-style-type: none"><li>인자를 매개변수로 복사해서 전달한다.</li><li>함수 안에서 매개변수(복사본)를 변경해도 인자(원본)은 변경되지 않는다.</li></ul>
참조에 의한 호출 (참조 호출)	<ul style="list-style-type: none"><li>인자의 주소를 포인터형의 매개변수로 전달한다.</li><li>함수 안에서 매개변수(참조)가 가리키는 인자(원본)을 변경할 수 있다.</li></ul>

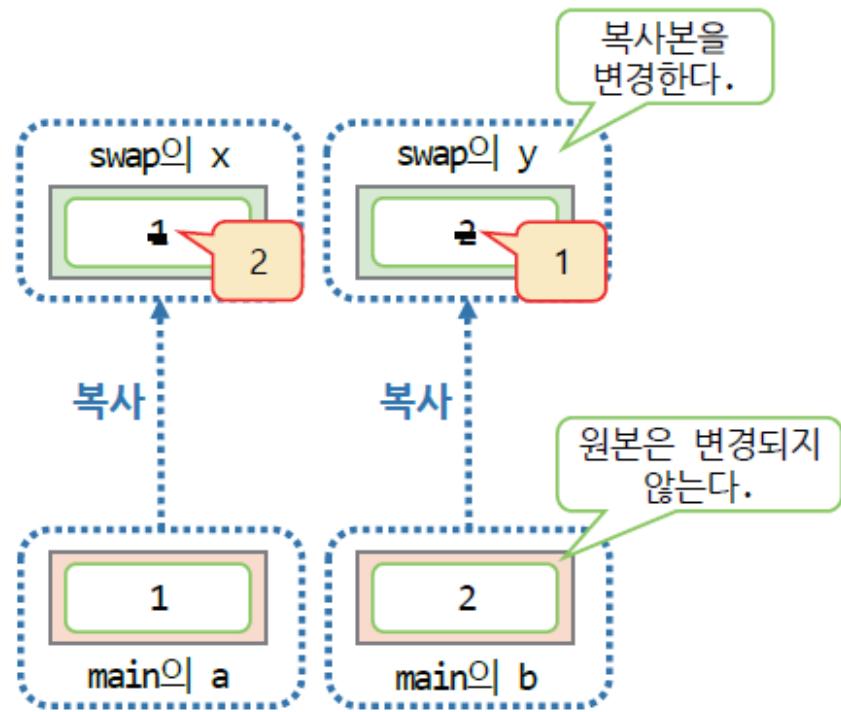
# 함수와 포인터\_함수의 인자 전달 방법

## ❖ 값에 의한 호출

- 인자를 매개변수로 복사해서 전달한다.
  - 함수의 매개변수는 함수가 호출될 때 생성되는 지역 변수로, 인자의 값으로 초기화된다.

### 값에 의한 호출

```
void swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp
}
int main(void)
{
    int a = 1, b = 2;
    swap(a, b);
}
```



swap 함수는 값에 의한 호출로는 구현할 수 없다.

# 함수와 포인터\_함수의 인자 전달 방법

## ❖ 참조에 의한 호출

- 변수의 복사본을 전달하는 대신 변수에 대한 참조를 전달한다.
  - 함수 안에서 직접 변수를 변경할 수 있다.
  - C에서는 포인터를 이용해서 참조에 의한 호출을 처리한다.
- 매개변수는 인자를 가리키는 포인터로 선언한다.

```
void swap(int* px, int* py);
```

- 함수를 호출할 때는 인자로 전달하려는 변수의 주소를 전달한다.

```
swap(&a, &b);
```

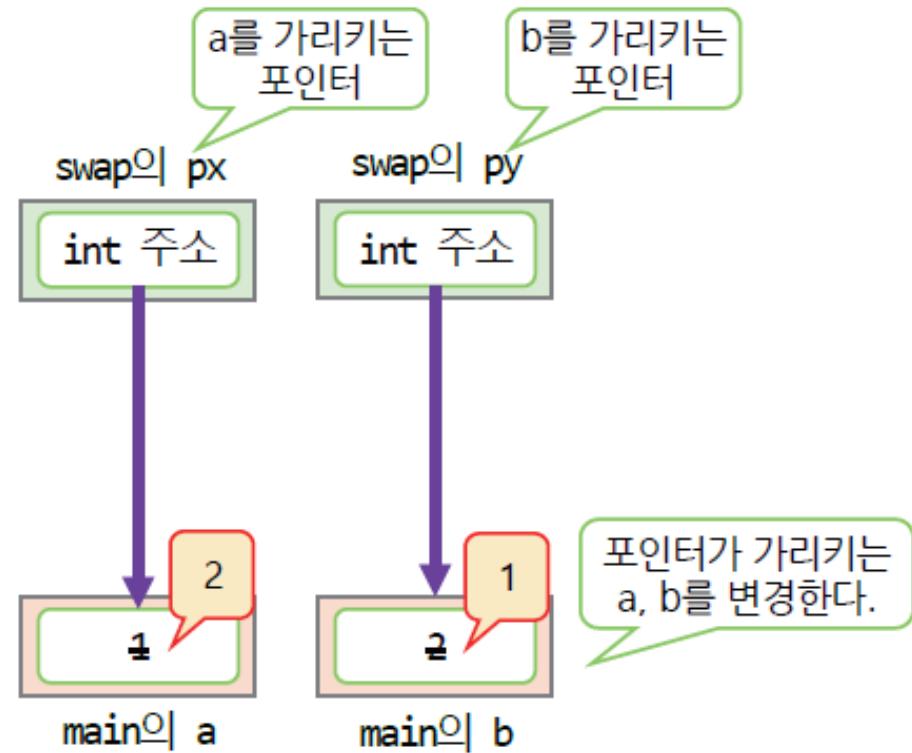
# 함수와 포인터\_함수의 인자 전달 방법

## ❖ 참조에 의한 호출

- 함수를 정의할 때는 포인터형의 매개변수를 역참조해서 매개변수가 가리키는 변수에 접근한다.

### 참조에 의한 호출

```
void swap(int *px, int *py)
{
    int temp = *px;
    *px = *py;
    *py = temp;
}
int main(void)
{
    int a = 1, b = 2;
    swap(&a, &b);
}
```



# 함수와 포인터\_함수의 인자 전달 방법

## ❖ swap함수의 구현(1/2)

```
01 #include <stdio.h>
02 void swap(int* px, int* py); } ----- 매개변수는 인자에 대한
03
04 int main(void)
05 {
06     int a = 1, b = 2;
07
08     printf("a = %d, b = %d\n", a, b);
09     swap(&a, &b); } ----- 인자로 전달하려는 변수
10     printf("a = %d, b = %d\n", a, b);
11 }
```

매개변수는 인자에 대한  
포인터형으로 선언한다.

인자로 전달하려는 변수  
의 주소를 전달한다.

# 함수와 포인터\_함수의 인자 전달 방법

## ❖ swap함수의 구현(2/2)

```
12  
13 void swap(int* px, int* py)  
14 {  
15     int temp = *px;  
16     *px = *py;  
17     *py = temp;  
18 }
```

매개변수가 가리키는  
변수를 변경한다.

a, b의 값이  
서로 바뀐다.

실행 결과

a = 1, b = 2  
a = 2, b = 1

# 함수와 포인터\_함수의 인자 전달 방법

## ❖ 매개변수의 역할에 따른 인자 전달 방법 결정

구분	역할	인자 전달 방법
입력 매개변수	<ul style="list-style-type: none"><li>함수를 호출한 곳에서 입력을 받아 오기 위한 매개변수</li><li>함수 안에서 값이 사용될 뿐 변경되지 않는다.</li></ul> <pre>int add(int x, int y);</pre>	값에 의한 호출
출력 매개변수	<ul style="list-style-type: none"><li>함수의 출력을 함수를 호출한 곳으로 전달하기 위한 매개변수</li><li>함수 안에서 변경된다.</li><li>출력 매개변수는 포인터로 전달한다.</li></ul> <pre>void get_sum_product(int x, int y, int *sum, int *product);</pre>	참조에 의한 호출
입출력 매개변수	<ul style="list-style-type: none"><li>함수의 입력과 출력 모두로 사용되는 매개변수</li><li>함수 안에서 그 값이 사용도 되고 변경도 된다.</li><li>입출력 매개변수도 포인터로 전달한다.</li></ul> <pre>void swap(int* px, int* py);</pre>	

- 함수의 처리 결과가 2개 이상인 경우에 출력 매개변수를 사용한다.

# 함수와 포인터\_함수의 인자 전달 방법

## ❖ 함수의 처리 결과를 출력 매개변수로 전달하는 방법 (1/2)

- 함수의 원형을 정할 때는 출력 매개변수를 포인터로 선언한다.
  - 처리 결과를 저장할 변수를 가리키는 포인터형으로 선언한다.

```
void get_sum_product(int x, int y, int *psum, int *pproduct);
```

x, y의 합과 곱을  
구하는 함수

- 함수를 호출할 때는, 처리 결과를 받아올 변수의 주소를 전달한다.

```
int sum, product;
```

처리 결과를 받아올  
변수를 준비한다.

```
get_sum_product(123, 456, &sum, &product);
```

# 함수와 포인터\_함수의 인자 전달 방법

## ❖ 함수의 처리 결과를 출력 매개변수로 전달하는 방법 (2/2)

- 함수를 정의할 때는, 포인터형의 매개변수가 가리키는 곳에 처리 결과를 저장한다.

```
void get_sum_product(int x, int y, int *psum, int *pproduct)
{
    *psum = x + y;          // psum이 가리키는 변수에 합을 저장한다.
    *pproduct = x * y;      // pproduct가 가리키는 변수에 곱을 저장한다.
}
```

# 함수와 포인터\_함수의 인자 전달 방법

## ❖ 출력 매개변수를 갖는 함수

```
01 #include <stdio.h>
03 void get_sum_product(int x, int y, int *psum, int *pproduct);
05 int main(void)
06 {
07     int sum, product;
10     get_sum_product(123, 456, &sum, &product);
11     printf("sum = %d, product = %d\n", sum, product);
12 }
14 void get_sum_product(int x, int y, int *psum, int *pproduct)
15 {
17     *psum = x + y;
18     *pproduct = x * y;
19 }
```

실행 결과

sum = 579, product = 56088

# 함수와 포인터\_배열의 전달

## ❖ 배열은 항상 포인터로 전달한다.

- 배열을 값에 의한 호출로 전달하면 배열 전체를 복사해야 하므로 시간적, 공간적 성능 저하가 발생한다.

```
void print_array(int arr[], int size);  
void print_array(int *arr, int size);
```

두 문장은 항상 같은 뜻이다.

## ❖ 배열의 크기는 별도의 매개변수로 받아와야 한다.

- *arr*는 포인터이므로 *arr*로 배열의 크기를 구할 수 없기 때문

## ❖ 함수 호출 시 배열을 전달하려면, 배열 이름, 즉 배열의 시작 주소를 전달한다.

```
int x[SIZE] = { 10, 20, 30, 40, 50 };
```

```
print_array(x, SIZE);
```

배열의 시작 주소 *x*를 *arr*로 전달한다.

# 함수와 포인터\_배열의 전달

❖ 함수 안에서는 포인터형의 매개변수를 배열처럼 사용한다.

```
void print_array(int *arr, int size)
{
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
}
```

arr는 포인터지만 배열처럼 사용한다.

arr[i]는 \*(arr+i)로 처리된다.

❖ 배열이 입력 매개변수일 때는 읽기 전용 포인터로 선언하는 것이 좋다.

- const 포인터로 선언하면 함수 안에서 arr 가 가리키는 배열의 원소를 변경할 수 없다.

```
void copy_array(const int *source, int *target, int size)
```

입력 매개변수

출력 매개변수

# 함수와 포인터\_배열의 전달

## ❖ 배열을 매개변수로 갖는 함수(1/3)

```
01 #include <stdio.h>
02 #define SIZE 5
03
04 void copy_array(const int *source, int *target, int size);
05 void print_array(const int *arr, int size);
06
07 int main(void)
08 {
09     int x[SIZE] = { 10, 20, 30, 40, 50 };
10     int y[SIZE] = { 0 };
11
12     printf("x = ");
13     print_array(x, SIZE);
```

배열의 시작 주소 x를 arr로 전달한다.

# 함수와 포인터\_배열의 전달

## ❖ 배열을 매개변수로 갖는 함수(2/3)

```
15     copy_array(x, y, SIZE);          배열의 시작 주소를 함수의 인  
16     printf("y = ");                자로 전달한다.  
17     print_array(y, SIZE);  
18 }  
19  
21 void copy_array(const int *source, int *target, int size)    source는 입력 매개변수 target  
22 {  
23     int i;  
24     for (i = 0; i < size; i++)  
25         target[i] = source[i];        source, target은 포인터지만  
26 }  
27
```

source는 입력 매개변수 target은 출력 매개변수

source, target은 포인터지만 배열처럼 사용한다.

# 함수와 포인터\_배열의 전달

## ❖ 배열을 매개변수로 갖는 함수(3/3)

```
28 void print_array(const int *arr, int size)
```

```
29 {
```

```
30     int i;
```

```
31     for (i = 0; i < size; i++)
```

```
32         printf("%d ", arr[i]); }
```

```
33     printf("\n");
```

```
34 }
```

arr는 입력 매개변수

arr는 포인터지만 배열처럼  
사용한다.

x = 10 20 30 40 50

y = 10 20 30 40 50

# 함수와 포인터\_배열의 전달

## ❖ 배열을 함수의 인자로 전달하는 방법(1/2)

- 함수 원형을 정할 때, 함수의 매개변수는 배열의 크기를 생략하고 선언하거나 배열 원소에 대한 포인터형으로 선언

```
void print_array(int arr[]);
```

```
void print_array(int *arr);
```

- 함수 안에서 배열의 크기가 필요하면 배열의 크기도 매개변수로 받아와야 함

```
void print_array(int *arr, int size);
```

- 배열이 입력 매개변수일 때는 **const**를 지정

```
void print_array(const int *arr, int size);
```

# 함수와 포인터\_배열의 전달

## ❖ 배열을 함수의 인자로 전달하는 방법(2/2)

- 함수를 호출할 때는 배열 이름을 인자로 전달

```
int x[5] = { 10, 20, 30, 40, 50 };
print_array(x, 5);
```

- 함수를 정의할 때는 매개변수인 포인터를 배열처럼 사용

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