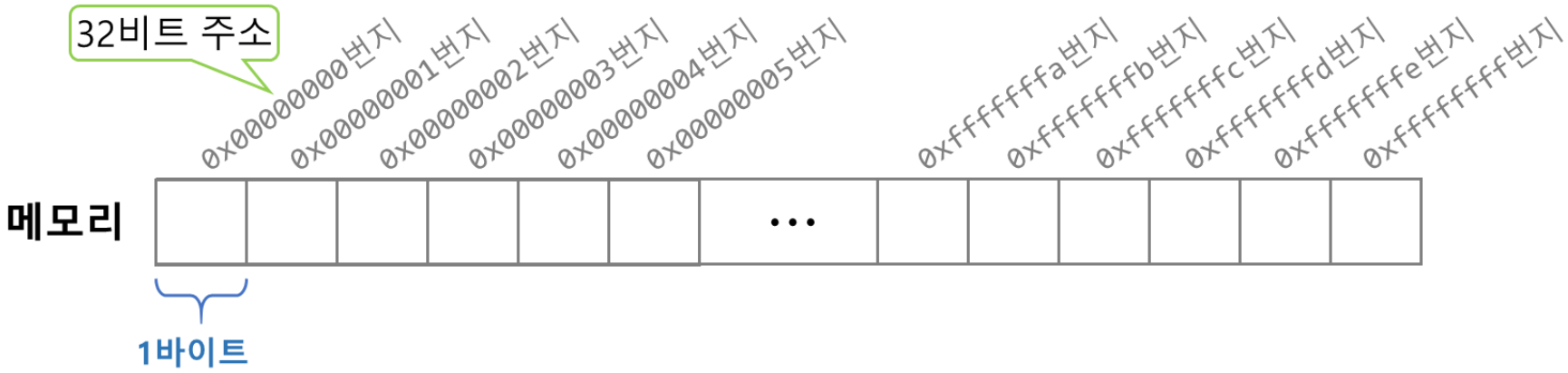
포인터

목차

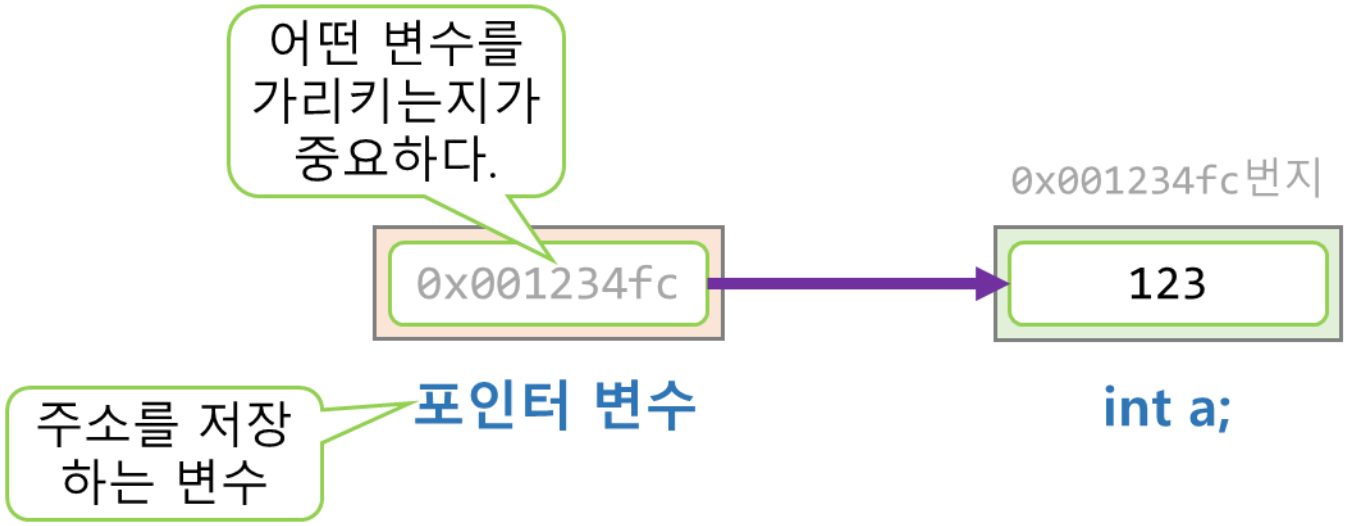
포인터의 개념 (1/2)

- 포인터(pointer)는 주소(address)를 저장하는 변수이다.
- 메모리에는 각각의 바이트를 구분하기 위한 주소(번지)가 있다.
  - 주소의 크기도 플랫폼에 따라 다르다.
  - 32비트 플랫폼에서는 주소가 4바이트이고, 64비트 플랫폼에서는 주소가 8바이트이다.



포인터의 개념 (2/2)

- 포인터를 사용할 때는 주소 값이 아니라 포인터가 어떤 변수를 가리키는지가 중요하다.
- 포인터는 다른 변수를 가리키는 변수이다.
  - 포인터는 주소를 이용해서 다른 변수에 접근할 수 있도록 도와준다.



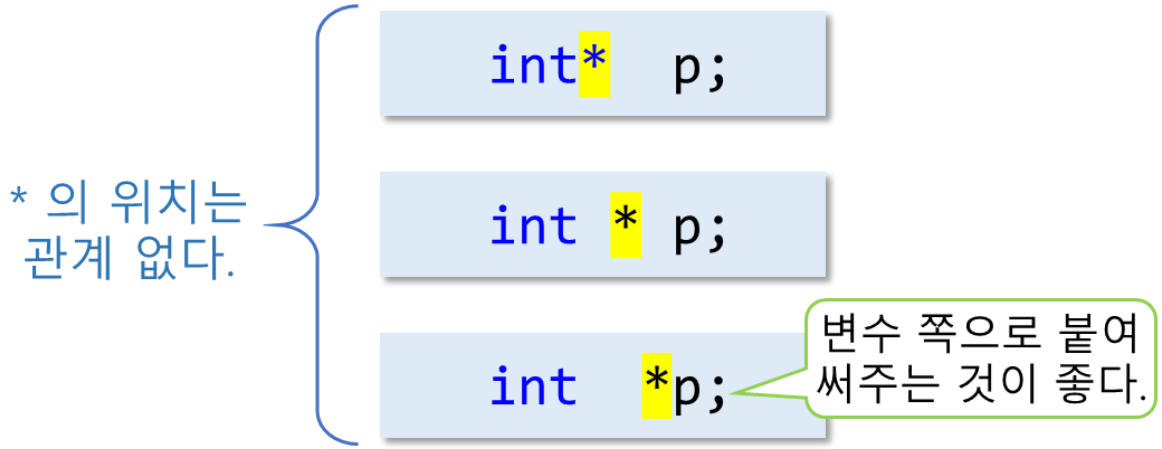
포인터의 선언 (1/2)

형식

데이터형 *변수명;
데이터형 *변수명 = 초기값;

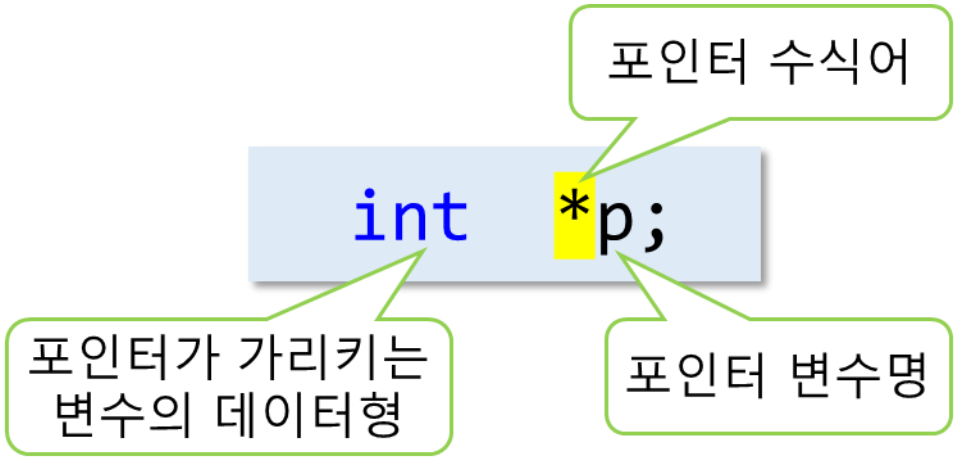
사용예

```
int *p;
double *pd;
int a = 123;
int *pa = &a;
char *pc = NULL;
```

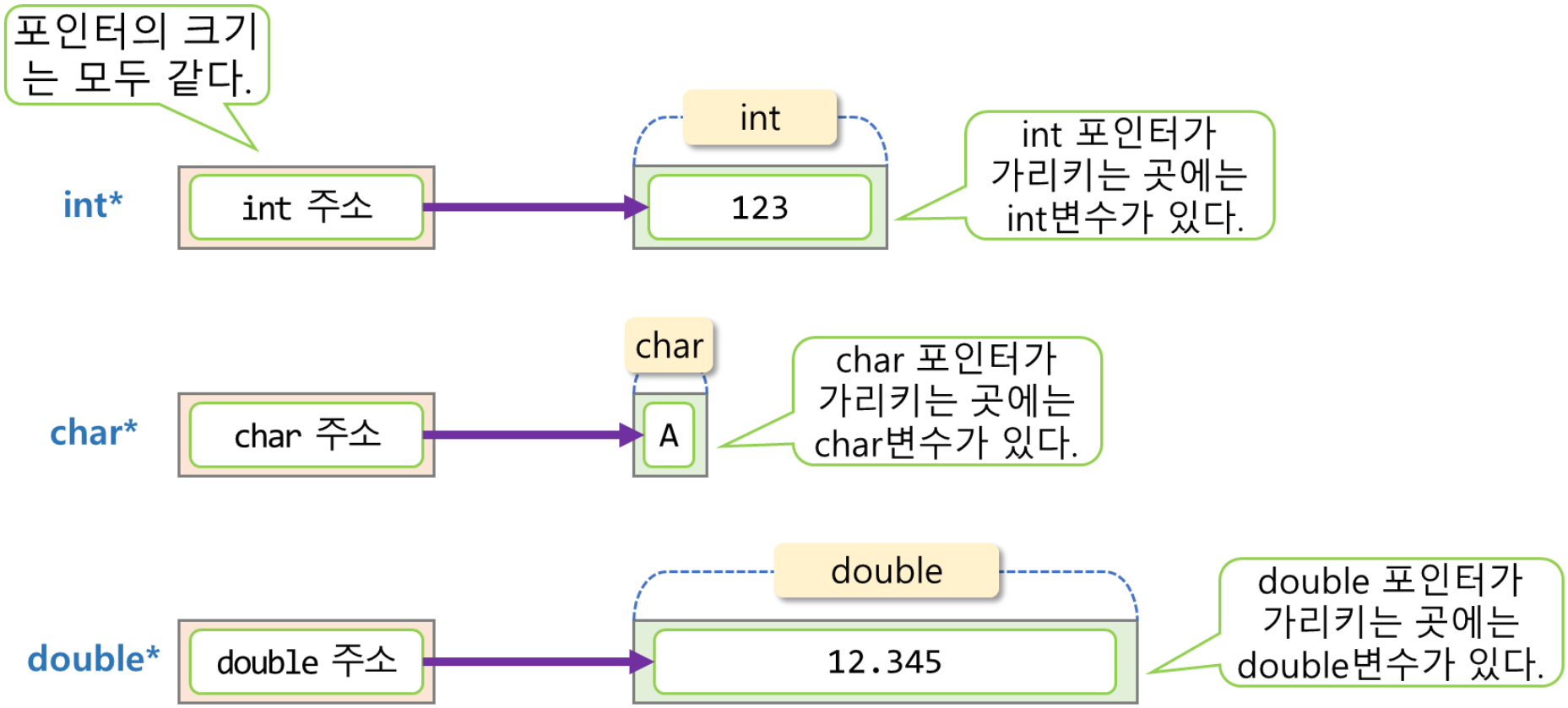


포인터의 선언 (2/2)

- 포인터를 선언할 때 지정하는 데이터형은 포인터가 가리키는 변수의 데이터형이다.



포인터의 의미


포인터의 크기는 모두 같다.
int*
int 주소
int
123
int 포인터가 가리키는 곳에는 int변수가 있다.
char*
char 주소
char
A
char 포인터가 가리키는 곳에는 char변수가 있다.
double*
double 주소
double
12.345
double 포인터가 가리키는 곳에는 double변수가 있다.

포인터의 크기

- 포인터의 데이터형이 다르더라도 포인터의 크기는 항상 같다.
  - 포인터의 크기는 플랫폼에 의해서 결정된다.

```
int *pi;
double *pd;
printf("sizeof(pi) = %d\n", sizeof(pi));        // 4바이트 (32비트 플랫폼 기준)
printf("sizeof(pd) = %d\n", sizeof(pd));        // 4바이트
```

```
printf("sizeof(int*) = %d\n", sizeof(int*));              // 4바이트
printf("sizeof(double*) = %d\n", sizeof(double*));        // 4바이트
```

예제 8-1 : 포인터의 바이트 크기 구하기

```
03  int main(void)
04  {
05      int *pi;        // *는 변수명 쪽으로 붙여준다.
06      double *pd;
07      char *pc;
08
09      printf("sizeof(pi) = %zd\n", sizeof(pi));     // 4바이트 (32비트 플랫폼)
10      printf("sizeof(pd) = %zd\n", sizeof(pd));     // 4바이트
11      printf("sizeof(pc) = %zd\n", sizeof(pc));     // 4바이트
12
13      printf("sizeof(int*) = %zd\n", sizeof(int*));
14      printf("sizeof(double*) = %zd\n", sizeof(double*));
15      printf("sizeof(char*) = %zd\n", sizeof(char*));
16
17      return 0;
18  }
```

실행결과

```
sizeof(pi) = 4
sizeof(pd) = 4
sizeof(pc) = 4
sizeof(int*) = 4
sizeof(double*) = 4
sizeof(char*) = 4
```

포인터의 초기화 (1/2)

- 포인터에 직접 절대 주소를 대입해서는 안된다.

```
int *p2 = (int*)0x12345678;   // 메모리 주소를 직접 사용하면 실행 에러가 발생한다.
```

- 변수의 주소를 구할 때는 주소 구하기 연산자인 &를 이용한다.

```
int a = 10;
int *p3 = &a;        // int 변수 a의 주소를 구해서 int 포인터인 p3를 초기화한다.
```

포인터의 초기화 (2/2)

- 어떤 변수의 주소로 초기화할지 알 수 없으면 0으로 초기화한다.

```
int *p4 = 0;        // 어떤 변수의 주소로 초기화할지 알 수 없으면 0으로 초기화한다.
```

```
int *p5 = NULL;     // 0 대신 NULL을 사용해도 된다.
```

예제 8-2 : 포인터의 선언 및 초기화

```
03  int main(void)
04  {
05      //int *p1 = 0x12345678;           // 컴파일 에러
06      int *p2 = (int*)0x12345678;       // 실행 에러가 발생할 수 있다.
07
08      int a = 10;
09      int *p3 = &a;   // a의 주소를 구해서 p를 초기화한다.
10
11      int *p4 = 0;    // 어떤 주소로 초기화할지 알 수 없으면 0으로 초기화한다.
12      int *p5 = NULL; // 0 대신 NULL을 사용해도 된다.
13
14      printf("p2 = %p\n", p2);
15      printf("p3 = %p\n", p3);
16      printf("p4 = %p\n", p4);
17      printf("p4 = %p\n", p5);
18
19      return 0;
20  }
```

주소를 출력할 때는
%p 형식 문자열을 이용한다.

실행결과

```
p2 = 12345678
p3 = 00EFFA38
p4 = 00000000
p4 = 00000000
```

포인터의 사용 (1/2)

- 주소 구하기 연산자 &
  - 변수(l-value)에만 사용할 수 있다.

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

  - 상수나 수식에는 사용할 수 없다.

```
p = &123;       // ERROR
p = &(x + 1);   // ERROR
p = &printf("hello"); // ERROR
```

포인터의 사용 (2/2)

- 역참조(간접 참조) 연산자 *
  - 포인터가 가리키는 변수에 접근한다.

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

p가 가리키는 int 변수를 출력한다.

p가 가리키는 int 변수에 대입한다.

  - 포인터에만 사용할 수 있으며, 포인터가 아닌 변수나 수식에는 사용할 수 없다.

```
int x = 10;
*x = 30;        // ERROR
*(x + 1) = 40; // ERROR
```

x는 포인터가 아니므로 *를 사용할 수 없다.

수식에는 *를 사용할 수 없다.

역참조 연산의 의미

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

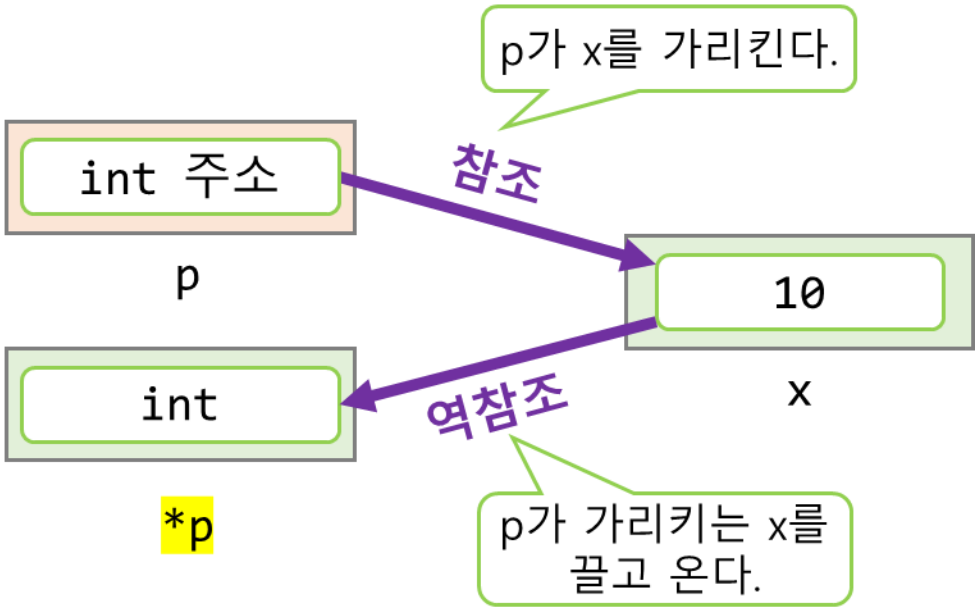
p가 x를 가리킨다.


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

x

x

*p가 사용되는 곳에 x가 대신 사용되는 것처럼 처리된다.



예제 8-3 : 포인터의 사용

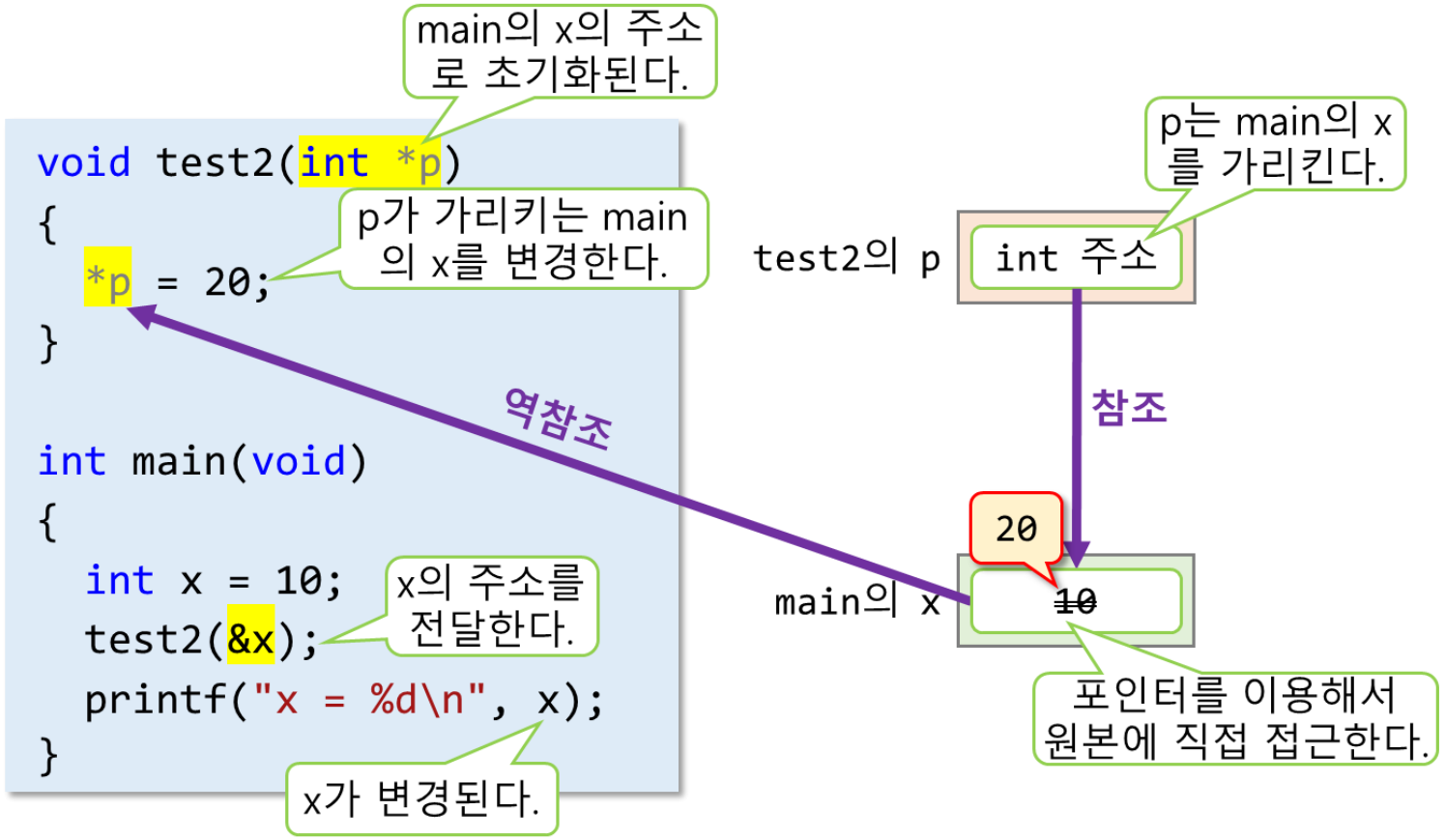
```
03  int main(void)
04  {
05      int x = 10;
06      int *p = &x;    // p는 a의 주소로 초기화한다.
07
08      printf(" x = %d\n", x);
09      printf("&x = %p\n", &x);  // &x는 주소 값이므로 %p로 출력
10
11      printf(" p = %p\n", p);
12      printf("*p = %d\n", *p);  // *p는 int형 변수이므로 %d로 출력
13      printf("&p = %p\n", &p);  // 포인터도 변수이므로 주소가 있다.
14
15      *p = 20;                  // x = 20;으로 수행된다.
16      printf("*p = %d\n", *p);  // printf("*p = %d\n", x);로 수행된다.
17
18      return 0;
19  }
```

실행결과

```
 x = 10
&x = 0117FA00
 p = 0117FA00
*p = 10
&p = 0117F9F4
*p = 20
```

포인터의 용도 (1/2)

- 변수의 이름을 직접 사용할 수 없을 때
  - 함수를 호출한 곳에 있는 지역 변수를 포인터를 이용해서 변경할 수 있다.



예제 8-4 : 포인터가 필요한 경우 (1/2)

```
03  void test1(int x)   // 매개변수 x는 main의 x로 초기화된 지역 변수
04  {
05      x = 20;         // x는 test1의 지역 변수이므로 test1이 리턴할 때 소멸된다.
06  }
07
08  void test2(int *p)  // p는 main의 x의 주소로 초기화된 포인터이다.
09  {
10      *p = 20;        // p가 가리키는 변수, 즉 main의 x에 20을 대입한다.
11  }
```

예제 8-4 : 포인터가 필요한 경우 (2/2)

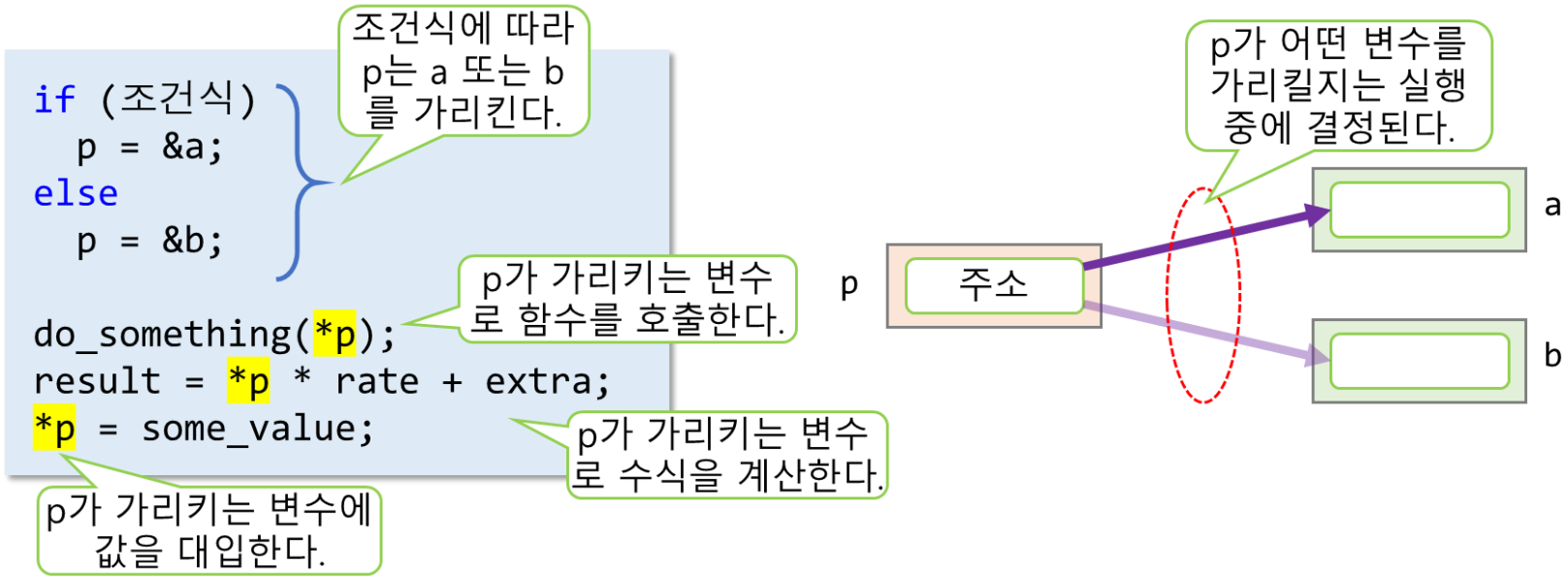
```
13  int main(void)
14  {
15      int x = 10;
16      test1(x);       // main의 x를 함수의 매개변수 x로 복사해서 전달한다.
17      printf("test1 호출 후 x = %d\n", x);          // x의 값은 변경되지 않는다.
18
19      test2(&x);      // test2 함수를 호출할 때 x의 주소를 넘겨준다.
20      printf("test2 호출 후 x = %d\n", x);          // x의 값이 변경된다.
21
22     return 0;
23  }
```

실행결과

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

포인터의 용도 (2/2)

- 포인터가 어떤 변수를 가리키게 될지 아직 모르는 경우
  - 포인터가 가리키는 변수가 프로그램 실행 중에 조건에 따라서 결정된다.
  - 여러 변수에 대한 처리를 공통의 코드로 수행하게 만들 수 있다.



포인터 사용 시 주의 사항 (1/3)

- 포인터는 초기화하고 사용하는 것이 안전하다.
  - 포인터를 초기화하지 않고 사용하면 실행 에러가 발생한다.

```
int *q;     // 쓰레기 값
*q = 10;    // 실행 에러 발생
```

  - 어떤 변수를 가리킬지 알 수 없으면 널 포인터로 초기화한다.

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

포인터 사용 시 주의 사항 (2/3)

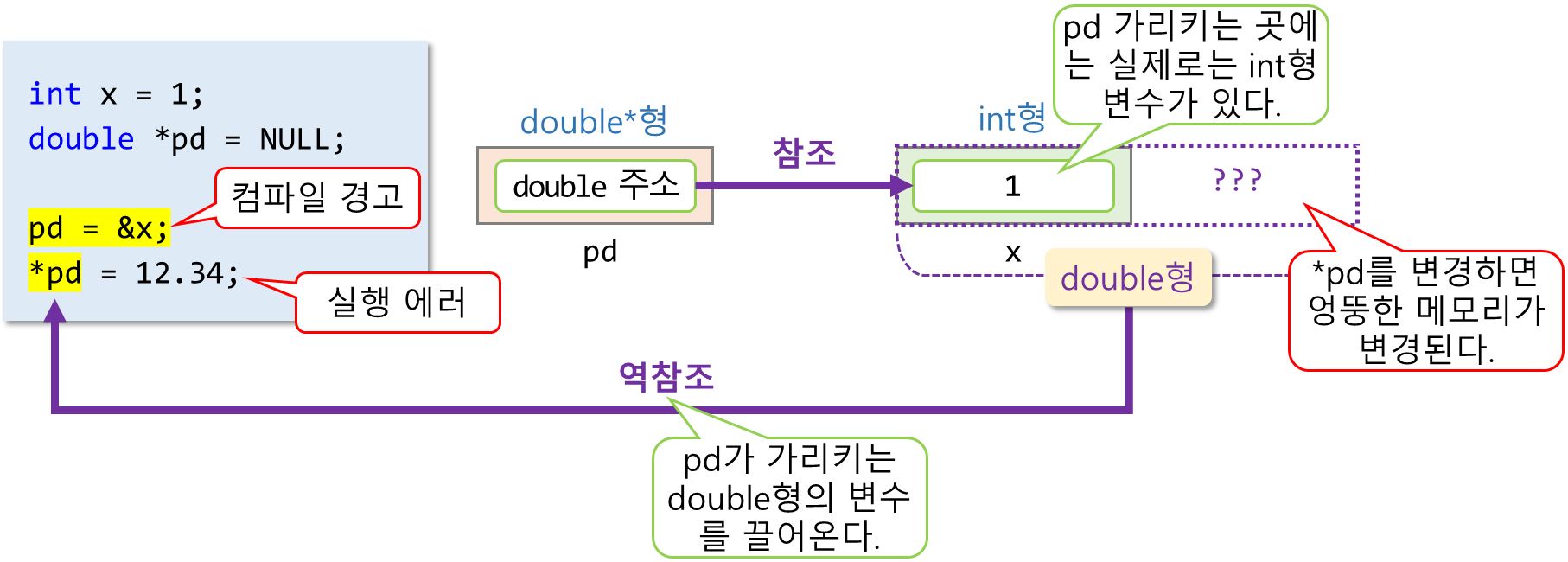
- 포인터를 안전하게 사용하려면 사용할 때 널 포인터인지 검사한다.
  - 널 포인터가 아닌 경우에만 포인터가 가리키는 변수에 접근한다.

```
if (q != NULL) // 널 포인터 검사
*q = 100;
```

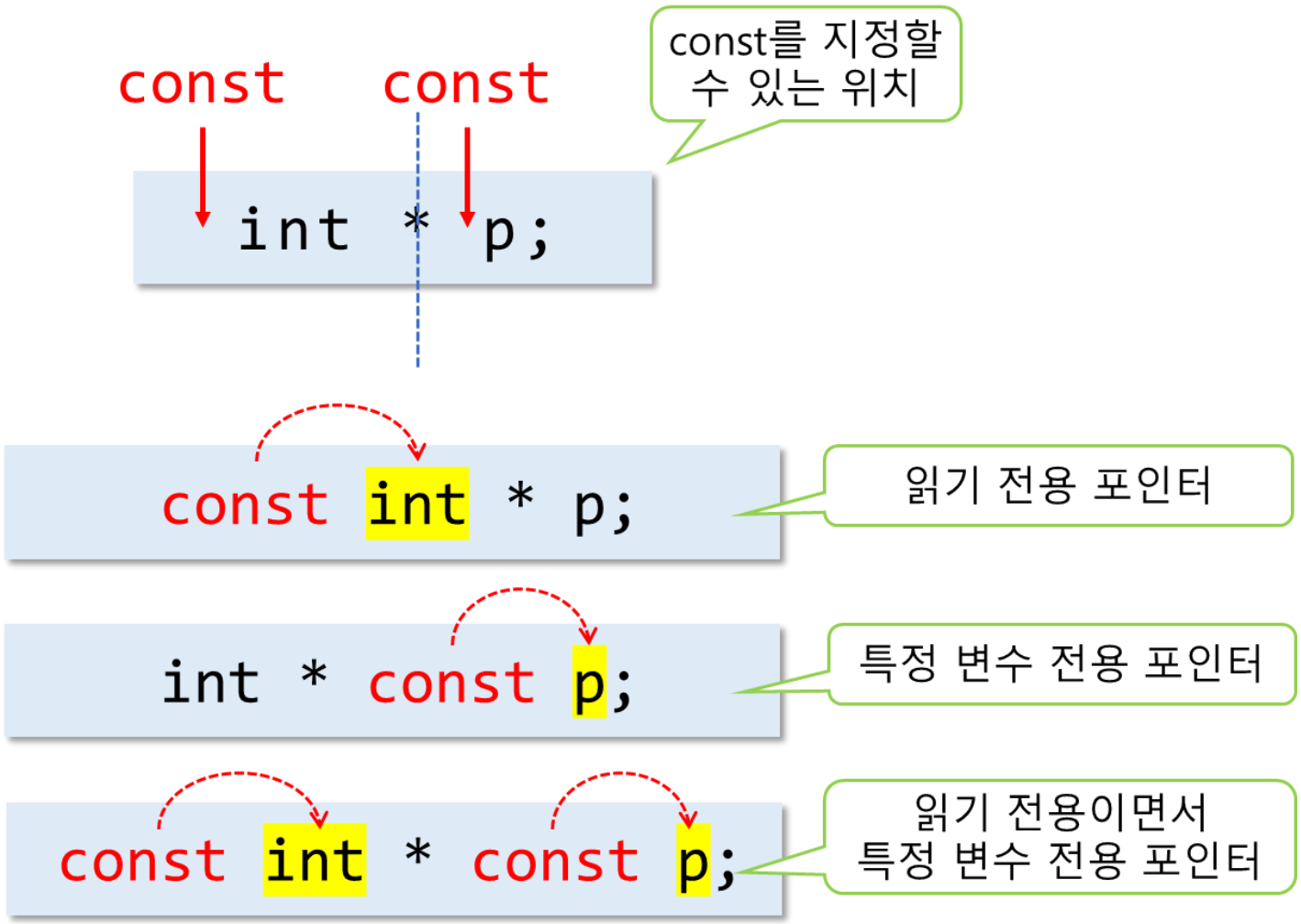
```
if (q)         // 널 포인터 검사
*q = 100;
```

포인터 사용 시 주의 사항 (3/3)

- 포인터의 데이터형과 포인터가 가리키는 변수의 데이터형이 같아야 한다.



const 포인터


const
const
int * p;
const를 지정할 수 있는 위치
const int * p;
읽기 전용 포인터
int * const p;
특정 변수 전용 포인터
const int * const p;
읽기 전용이면서 특정 변수 전용 포인터

const 데이터형 *변수; (1/2)

- 읽기 전용 포인터
  - 포인터가 가리키는 변수의 값을 변경할 수 없다.

```
int a = 10, b = 20;
const int *p1 = &a; // 읽기 전용 포인터

printf("*p1 = %d\n", *p1); // OK
*p1 = 100;       // 컴파일 에러
```

a를 직접 변경할 수는 있다.

```
a = 100;    // OK
(*p1)++;    // 컴파일 에러
```

p1으로 접근할 때는 변경할 수 없다.

const 데이터형 *변수; (2/2)

- 포인터 자신의 값(포인터에 저장된 주소)은 변경할 수 있다.
  - 포인터가 다른 변수를 가리키게 만들 수는 있다.

```
p1 = &b;                          // 이제 p1은 b를 가리킨다.
printf("*p1 = %d\n", *p1);        // p1이 가리키는 b를 출력한다.
```

- 선언 시 널 포인터로 초기화하고, 원하는 시점에 특정 변수의 주소를 저장하고 사용할 수 있다.

데이터형 * const 변수;

- 특정 변수의 전용 포인터
  - 포인터 자신의 값(포인터에 저장된 주소)을 변경할 수 없다.
  - 다른 변수를 가리킬 수 없다.

```
int *const p2 = &a;    // a 전용 포인터
p2 = &b;               // 컴파일 에러
```

  - 포인터가 가리키는 변수의 값은 변경할 수 있다.

```
*p2 = 100; // p2가 가리키는 변수의 값을 변경할 수 있다.
```

  - 선언 시 반드시 초기화해야 한다.

```
int *const p2; // 초기화하지 않으면 p2는 쓰레기 값이고
               // 나중에 주소를 저장할 수도
없다.
```

const 데이터형 * const 변수;

- 읽기 전용 포인터이면서 특정 변수 전용 포인터
  - 반드시 초기화해야 하며, 이 포인터로는 가리키는 변수의 값도 변경할 수 없고 포인터 자신의 값(포인터에 저장된 주소)도 변경할 수 없다.

```
const int * const p3 = &a;    // a 전용 포인터이면서
                              // a에 읽기 전용으로 접근
*p3 = 100;      // 컴파일 에러
p3 = &b;        // 컴파일 에러
```

예제 8-5 : const 포인터의 의미 (1/2)

```
03  int main(void)
04  {
05      int a = 10, b = 20;
06
07      const int *p1 = &a;       // p1는 a에 읽기 전용으로 접근한다.
08      int *const p2 = &a;       // p2는 a 전용 포인터이다.
09      const int * const p3 = &a;          // p3는 읽기 전용 + a 전용 포인터
10
11      printf("*p1 = %d\n", *p1);          // p1으로 a를 읽어 온다.
12      //*p1 = 100;    // *p1은 const 변수로 간주되므로 컴파일 에러
13      p1 = &b;        // p1이 다른 변수를 가리킬 수는 있다. 이제 p1은 b를 가리킨다.
14      printf("*p1 = %d\n", *p1);  // p1으로 b를 읽어 온다.
15
16      //p2 = &b;      // p2가 다른 변수를 가리키게 할 수 없으므로 컴파일 에러
17      *p2 = 100;      // p2가 가리키는 변수의 값을 변경할 수 있다.
18      printf("*p2 = %d\n", *p2);
```

예제 8-5 : const 포인터의 의미 (2/2)

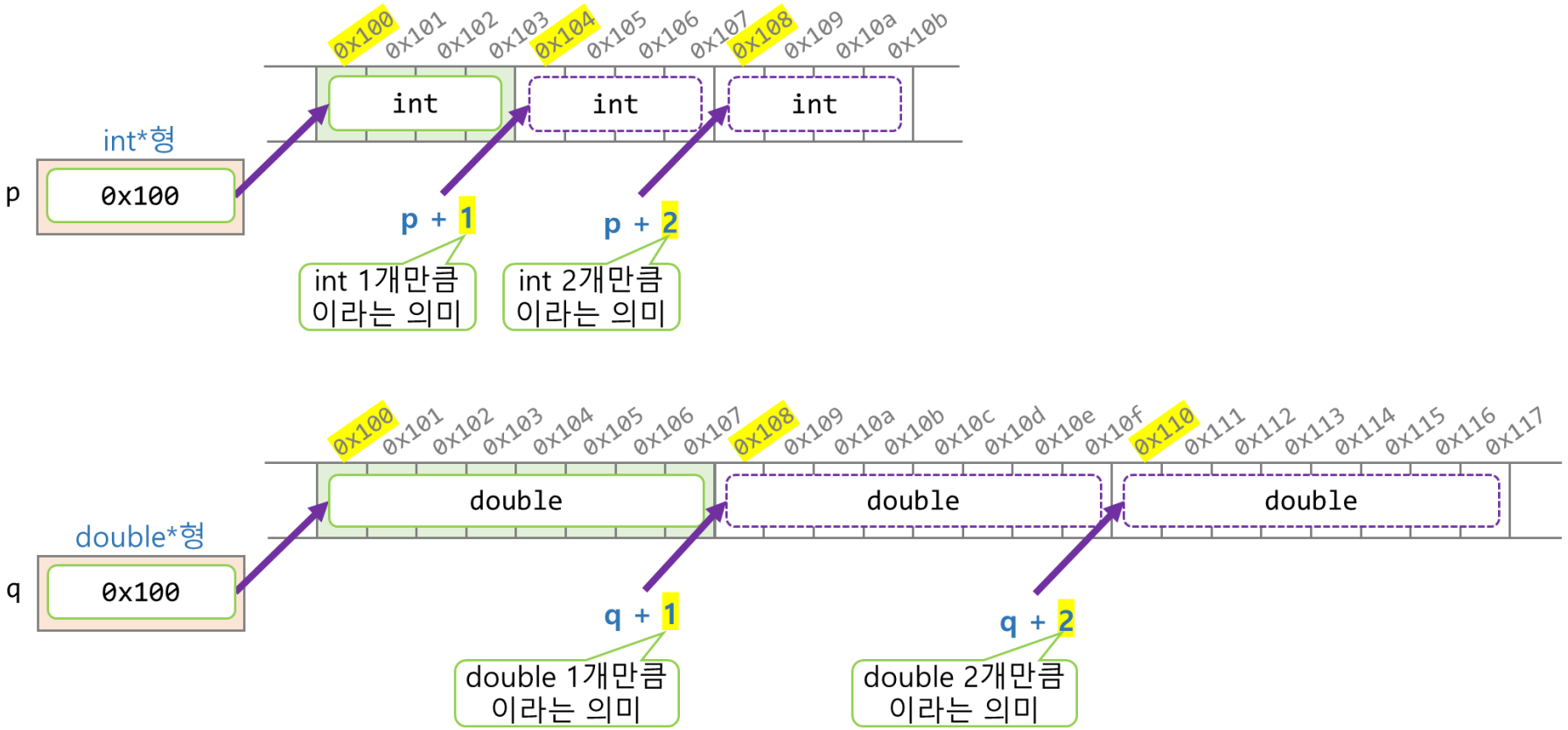
```
19
20      //*p3 = 100;     // 컴파일 에러
21      //p3 = &b;       // 컴파일 에러
22      printf("*p3 = %d\n", *p3);          // p3이 가리키는 변수의 값을 읽어 온다.
23
24      return 0;
25  }
```

실행결과

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

포인터와 +, – 연산 (1/2)

- p+N 연산의 결과는 p가 가리키는 데이터형 N 개 크기만큼 더한 주소이다.



예제 8-6 : '포인터+정수' 연산의 결과

```
03  int main(void)
04  {
05      int *p = (int*)0x100;      // 포인터 연산을 확인하기 위해 절대 주소를 대입한다.
06      double *q = (double*)0x100;
07      char *r = (char*)0x100;
08
09      printf("int*   : %p, %p, %p\n", p, p + 1, p + 2);        // 4바이트씩 차이
10      printf("double*: %p, %p, %p\n", q, q + 1, q + 2);        // 8바이트씩 차이
11      printf("char*  : %p, %p, %p\n", r, r + 1, r + 2);        // 1바이트씩 차이
12
13      return 0;
14  }
```

실행결과

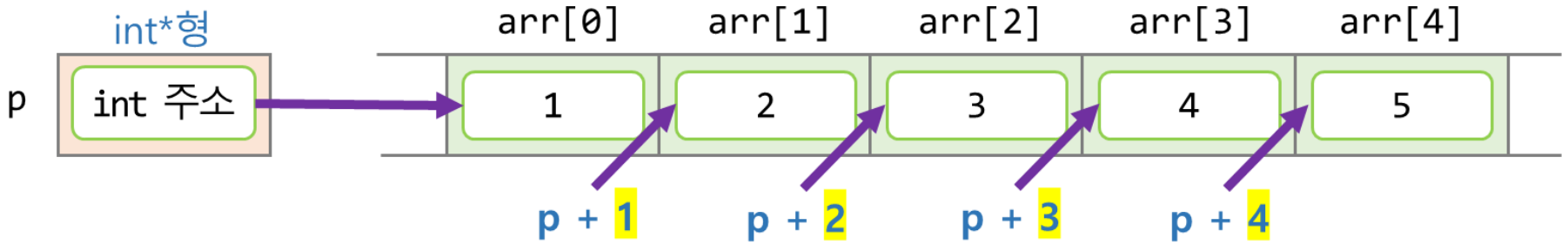
```
int*   : 00000100, 00000104, 00000108
double*: 00000100, 00000108, 00000110
char*  : 00000100, 00000101, 00000102
```

포인터와 +, - 연산 (2/2)

- '포인터+정수' 연산은 포인터가 가리키는 주소에 마치 배열이 있는 것처럼 메모리에 접근한다.

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

p가 arr[0]을 가리킬 때




```
p + i == &arr[i]
```

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

예제 8-7 : 배열의 0번 원소를 가리키는 포인터의 +, - 연산

```
03  int main(void)
04  {
05      int arr[5] = { 1, 2, 3, 4, 5 };
06      int *p = &arr[0];          // arr[0]의 주소를 p에 저장할 수 있다.
07      int i;
08
09      for (i = 0; i < 5; i++)
10      {
11          printf("p + %d = %p, ", i, p + i);
12          printf("*(p + %d) = %d\n", i, *(p + i));
13      }
14
15      return 0;
16  }
```

실행결과

```
p + 0 = 0019F7E8, *(p + 0) = 1
p + 1 = 0019F7EC, *(p + 1) = 2
p + 2 = 0019F7F0, *(p + 2) = 3
p + 3 = 0019F7F4, *(p + 3) = 4
p + 4 = 0019F7F8, *(p + 4) = 5
```

포인터와 ++, -- 연산 (1/3)

- p++, ++p
  - p가 가리키는 데이터형 1개 크기만큼 주소를 증가시킨다.
- p--, --p
  - p가 가리키는 데이터형 1개 크기만큼 주소를 감소시킨다.

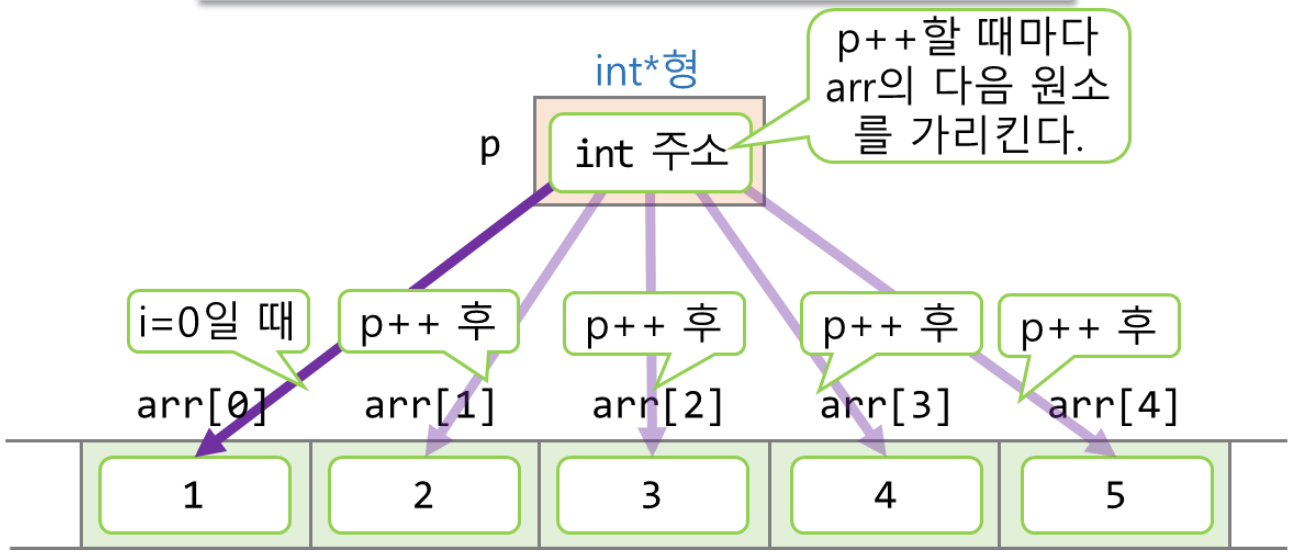
포인터와 ++, -- 연산 [2/3]

```
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 arr[0]을 가리킬 때

p가 가리키는 배열 원소를 출력한다.



예제 8-8 : 배열의 0번 원소를 가리키는 포인터와 증감 연산

```
03  int main(void)
04  {
05      int arr[5] = { 1, 2, 3, 4, 5 };
06      int *p = &arr[0];          // arr[0]의 주소를 p에 저장한다.
07      int i;
08
09      for (i = 0; i < 5; i++, p++)        // p는 &arr[0]~&arr[4]의 값이 된다.
10      {
11          printf("p= %p, ", p);           // p가 가리키는 원소가 계속 바뀐다.
12          printf("*p = %d\n", *p);        // p가 역참조하는 원소도 계속 바뀐다.
13      }
14
15      return 0;
16  }
```

실행결과

```
p= 00B3FAD4, *p = 1
p= 00B3FAD8, *p = 2
p= 00B3FADC, *p = 3
p= 00B3FAE0, *p = 4
p= 00B3FAE4, *p = 5
```

포인터와 ++, -- 연산 (3/3)

- 증감 연산자와 역참조 연산자를 함께 사용할 때는 연산자 우선순위를 신경써야 한다.
  - *p++은 *(p++)를 의미한다.

```
for (i = 0; i < 5; i++)
{
    printf("p= %p, ", p);
    printf("*p = %d\n", *p++);
}
```

*p를 출력한 다음 p++을 수행한다.

```
for (i = 0; i < 5; i++)
{
    printf("p= %p, ", p);
    printf("*p = %d\n", (*p)++);
}
```

p가 가리키는 arr[0]을 증가시킨다.
p는 변경되지 않는다.

배열처럼 사용되는 포인터 (1/3)

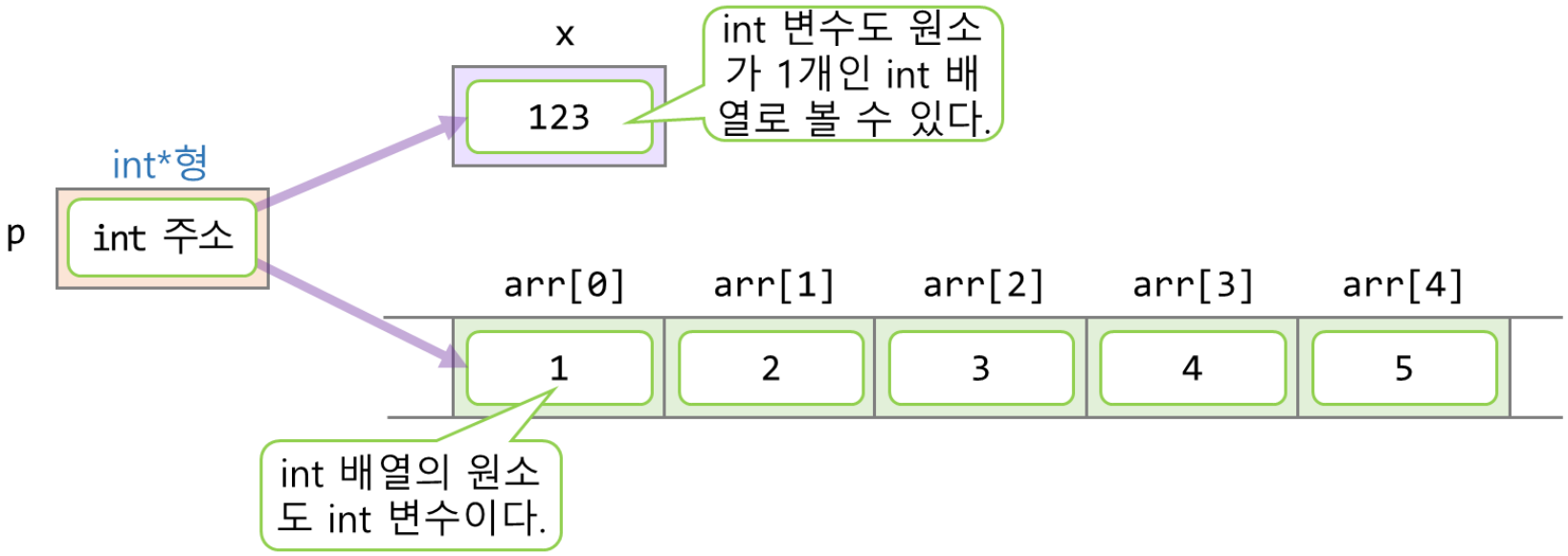
- 배열 원소를 가리키는 포인터
  - *type*형의 포인터는 항상 *type*형의 변수 또는 *type*형 배열의 원소를 가리킬 수 있다.

```
int x = 123;
int *p = &x;
```

int 변수를 가리킬 수 있다.

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

int 배열의 원소를 가리킬 수 있다.



배열처럼 사용되는 포인터 (2/3)

- 배열 원소를 가리키는 포인터는 배열 이름인 것처럼 사용할 수 있다.

```
for (i = 0; i < 5; i++)
    printf("%d ", p[i]);
```

p를 배열 이름인 것처럼 사용한다.

p가 가리키는 배열의 i번째 원소

p는 배열 원소를 가리키는 포인터

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

int 변수

```
p + i == &p[i]
```

주소

p가 가리키는 배열의 i번째 원소의 주소

배열처럼 사용되는 포인터 (3/3)

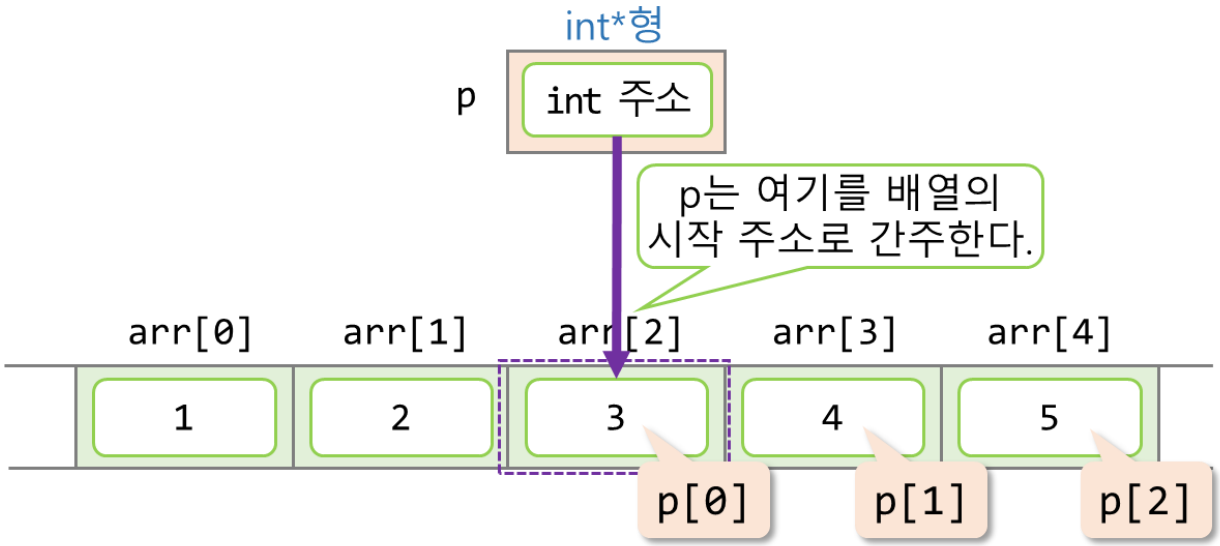
- 배열의 원소를 가리키는 포인터는 배열의 어떤 원소든지 가리킬 수 있다.

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

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

배열의 어떤 원소든지 가리킬 수 있다.

arr[2]~arr[4]를 의미한다.



예제 8-9 : 포인터를 배열인 것처럼 사용하는 경우

```
03  int main(void)
04  {
05      int arr[5] = { 1, 2, 3, 4, 5 };
06      int *p = arr;   // 배열의 이름, 배열의 시작 주소, &arr[0]은 모두 같다.
07      int i;
08
09      for (i = 0; i < 5; i++)
10          printf("p[%d] = %d\n", i, p[i]);          // p를 배열 이름인 것처럼 사용한다.
11      return 0;
12  }
```

실행결과

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

포인터처럼 사용되는 배열

- 배열의 이름은 배열의 시작 주소를 의미한다.
  - 배열 이름을 포인터인 것처럼 사용할 수 있다.

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

arr을 포인터인 것처럼 사용할 수 있다.

배열 이름을 포인터인 것처럼 사용할 수 있다.

arr는 배열의 이름

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

int 변수

```
&arr[i] == arr + i
```

주소

배열 vs. 포인터

- 배열 이름은 특정 변수 전용 포인터인 것처럼 사용할 수 있다.
- 배열의 시작 주소는 변경할 수 없다.

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

y = x;  // 컴파일 에러
x++;    // 컴파일 에러
```

배열의 시작 주소는 변경할 수 없다.

- 포인터는 값을 변경할 수 있으므로 포인터에 보관된 주소는 변경할 수 있다.

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

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

포인터에 저장된 주소는 다른 주소로 변경할 수 있다.

예제 8-10 : 배열과 포인터의 차이점 (1/2)

```
03  int main(void)
04  {
05      int x[5] = { 1, 2, 3, 4, 5 };
06      int y[5];
07      int *p = x;     // p는 x[0]을 가리킨다.
08      int i;
09
10      for (i = 0; i < 5; i++)
11          printf("%d ", p[i]);
12      printf("\n");
13
14      p = y;          // p는 이제 y[0]을 가리킨다.
15      for (i = 0; i < 5; i++)
16          p[i] = x[i];  // p가 가리키는 y 배열에 x 배열을 복사한다.
17
```

예제 8-10 : 배열과 포인터의 차이점 (2/2)

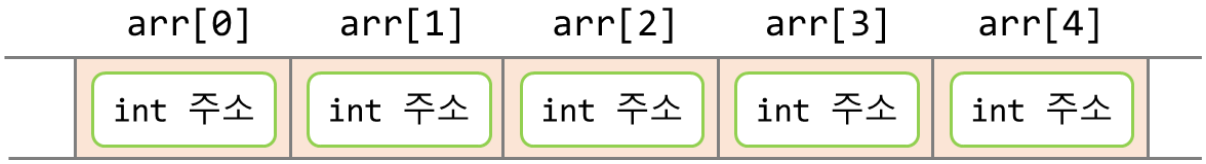
```
18      for (i = 0; i < 5; i++, p++)
19          printf("%d ", *p);
20      printf("\n");
21      return 0;
22  }
```

실행결과

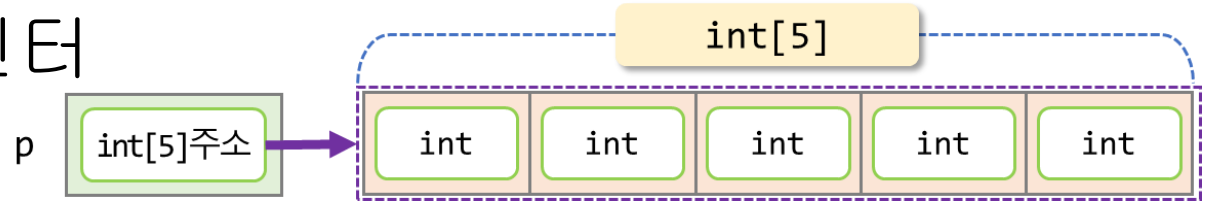
```
1 2 3 4 5
1 2 3 4 5
```

여러 가지 포인터의 선언

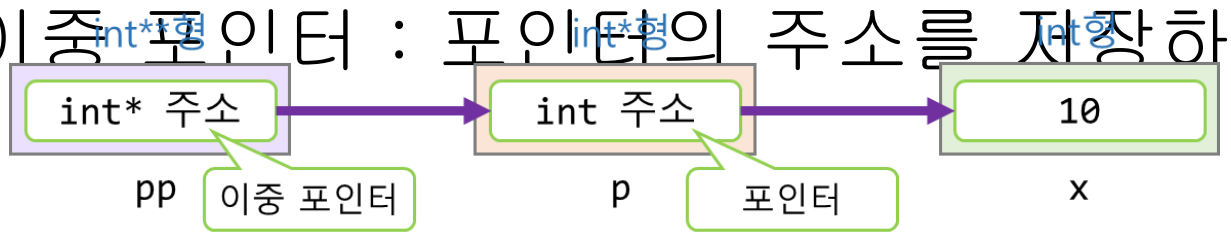
- 포인터 배열 : 주소를 저장하는 배열



- 배열에 대한 포인터 : 배열 전체를 가리키는 포인터



- 이중 포인터 : 포인터의 주소를 저장하는 포인터



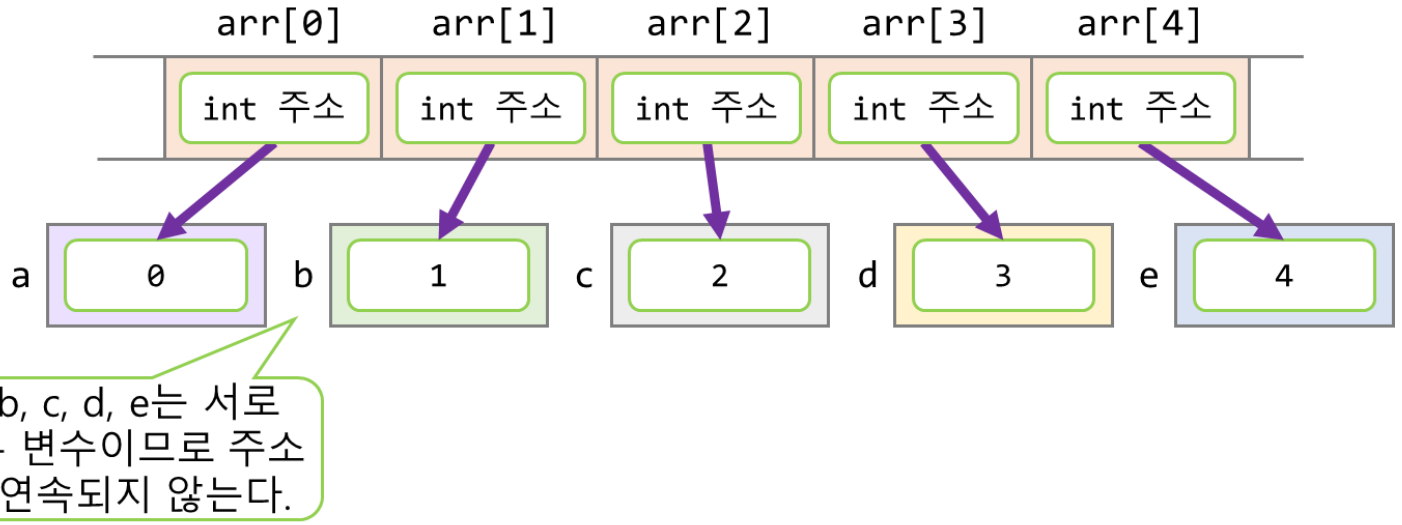
포인터 배열

- 포인터 배열의 각 원소가 다른 변수를 가리키는 포인터이다.

```
int a, b, c, d, e;
int *arr[5] = { &a, &b, &c, &d, &e };

for (i = 0; i < 5; i++)
    *arr[i] = i;
```

arr[i]가 가리키는 int형 변수



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

예제 8-11 : 포인터 배열의 선언 및 사용

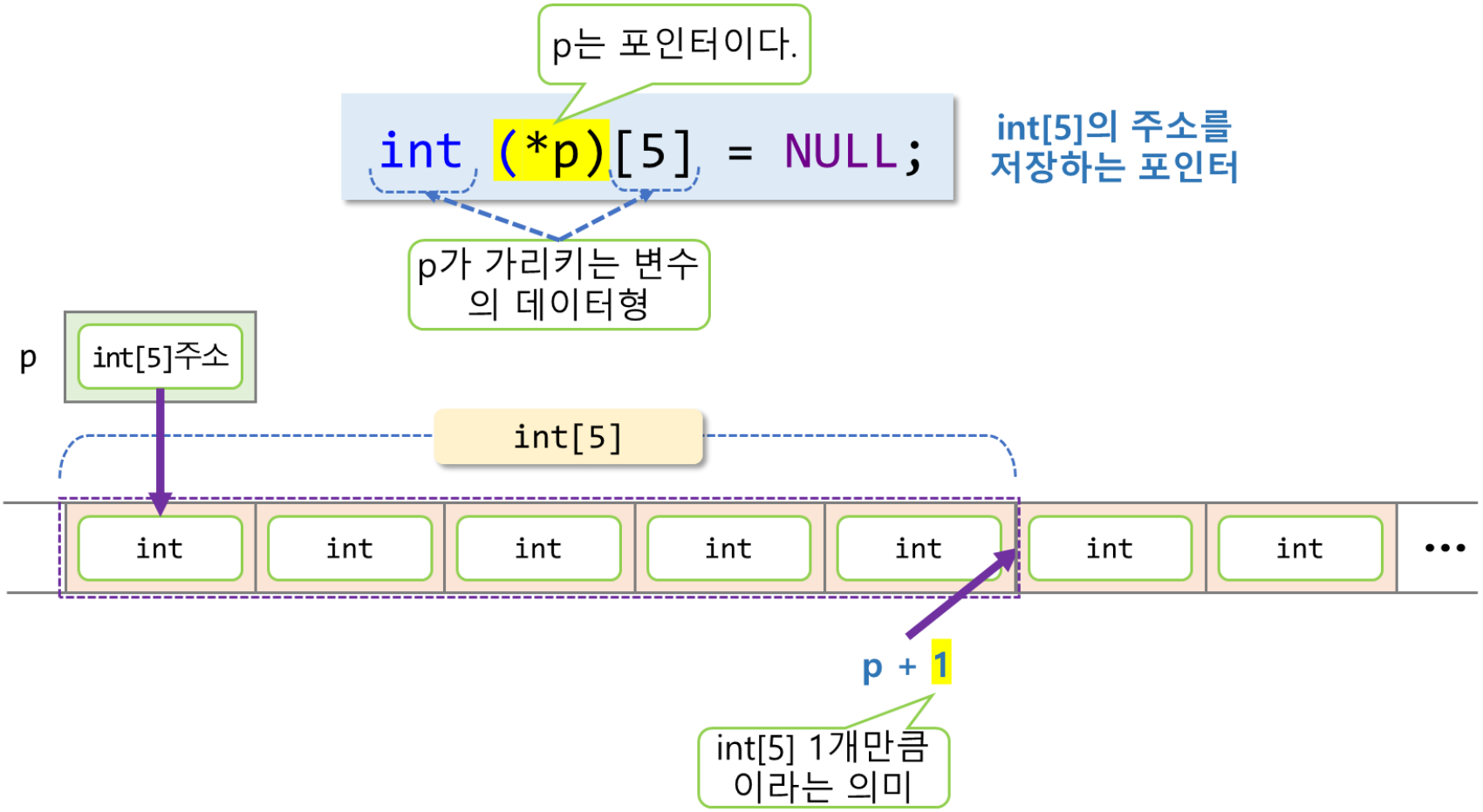
```
03  int main(void)
04  {
05      int a, b, c, d, e;
06      int *arr[5] = { &a, &b, &c, &d, &e };        // 포인터 배열
07      int i;
08
09      for (i = 0; i < 5; i++)
10      {
11          *arr[i] = i;
12          printf("%d ", *arr[i]);          // arr[i]는 포인터이다.
13      }
14      printf("\n");
15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

실행결과

```
0 1 2 3 4
```

배열에 대한 포인터 (1/2)

- 배열 전체를 가리키는 포인터



배열에 대한 포인터 (2/2)

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

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])

예제 8-12 : 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
```

이중 포인터

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

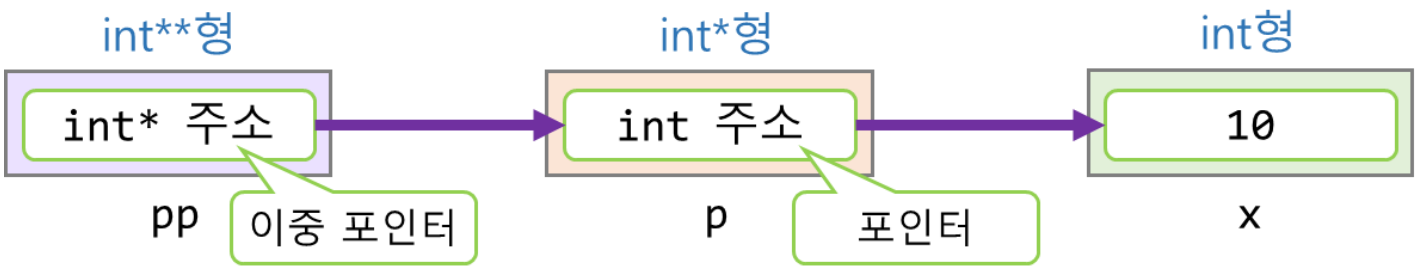
```
**pp = 20;
```
p
x

pp가 가리키는 변수의 데이터형

pp는 포인터 변수

```
int **pp = &p;
```

int*형 변수의 주소를 저장하는 포인터



함수의 인자 전달 방법

- 값에 의한 전달(passing by value)
  - 인자를 매개변수로 복사해서 전달하는 방식
  - 복사에 의한 전달
- 포인터에 의한 전달(passing by pointer)
  - 변수의 주소를 전달하는 방식
  - 함수를 호출한 곳에 있는 지역 변수의 주소를 매개변수로 받아오면 포인터를 통해서 해당 변수에 접근할 수 있다.
  - 함수의 처리 결과를 매개변수로 전달할 때 유용하게 사용

값에 의한 전달 : swap 함수

값에 의한 전달

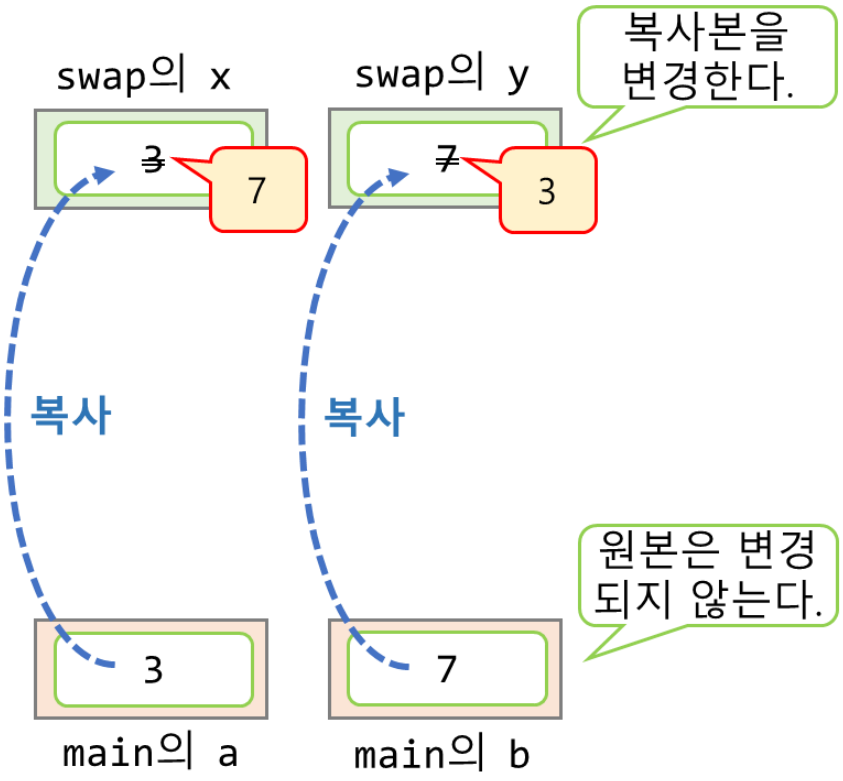
```
void swap(int x, int y)
{
   int temp = x;
   x = y;
   y = temp;
}

int main(void)
{
   int a = 3, b = 7;
   printf("a = %d, b = %d\n", a, b);

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

int x = a;

int y = b;



포인터에 의한 전달 : swap 함수

포인터에 의한 전달

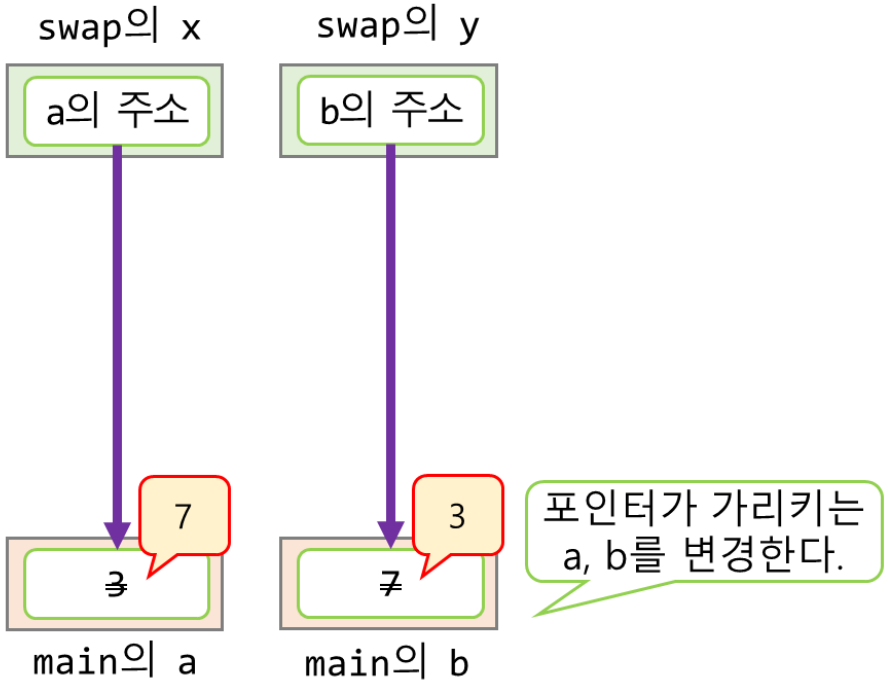
```
void swap(int *x, int *y)
{
   int temp = *x;
   *x = *y;
   *y = temp;
}

int main(void)
{
   int a = 3, b = 7;
   printf("a = %d, b = %d\n", a, b);

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

int *x = a;

int *y = b;



예제 8-13 : 포인터에 의한 전달 방법으로 구현한 swap 함수

```
04  int main(void)
05  {
06      int a = 3, b = 7;
07
08      printf("a = %d, b = %d\n", a, b);
09      swap(&a, &b);    // 포인터에 의한 전달
10      printf("a = %d, b = %d\n", a, b);
11      return 0;
12  }
13
14  void swap(int *x, int *y)     // x, y는 인자의 주소이다.
15  {
16      int temp = *x;  // x가 가리키는 변수의 값을 temp에 저장한다.
17      *x = *y;        // y가 가리키는 변수의 값을 x가 가리키는 변수에 저장한다.
18      *y = temp;      // temp를 y가 가리키는 변수에 저장한다.
19  }
```

실행결과

```
a = 3, b = 7
a = 7, b = 3
```

a, b의 값이 서로 바뀐다.

함수의 처리 결과를 매개변수로 전달하는 방법 (1/2)

- 함수의 원형을 정할 때, 처리 결과를 저장할 변수를 가리키는 포인터형으로 매개변수를 선언한다.

```
void get_sum_product(int x, int y, int *sum, int *product);
```

- 함수를 호출할 때, 처리 결과를 받아올 변수의 주소를 전달한다.

```
int result1, result2;
get_sum_product(10, 20, &result1, &result2);
```

함수의 처리 결과를 매개변수로 전달하는 방법 (2/2)

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

```
void get_sum_product(int x, int y, int *sum, int *product)
{
    *sum = x + y;
    *product = x * y;
}
```

예제 8-14 : 함수의 처리 결과를 매개변수로 전달하는 경우

```
04  int main(void)
05  {
06      int result1, result2;
07
08      // 2. 함수를 호출할 때 처리 결과를 받아올 변수의 주소를 전달한다
09      get_sum_product(10, 20, &result1, &result2);
10      printf("sum = %d, product = %d\n", result1, result2);
11      return 0;
12  }
13
14  // 1. 처리 결과를 저장할 변수를 가리키는 포인터형으로 매개변수를 선언한다.
15  void get_sum_product(int x, int y, int *sum, int *product)
16  {
17      // 3. 포인터형의 매개변수가 가리키는 곳에 처리 결과를 저장한다.
18      *sum = x + y;
19      *product = x * y;
20  }
```

실행결과

```
sum = 30, product = 200
```

함수의 매개변수

- 입력 매개변수(in parameter)
  - 함수를 호출한 곳에서 입력을 받아 오기 위한 매개변수
  - 함수 안에서 사용될 뿐 변경되지 않는다.
  - 입력 매개변수는 값으로 전달한다.
- 출력 매개변수(out parameter)
  - 함수의 출력을 함수를 호출한 곳으로 전달하기 위한 매개변수
  - 함수 안에서 변경된다.
  - 출력 매개변수는 포인터로 전달한다.
- 입출력 매개변수(in-out parameter)
  - 함수의 입력과 출력 모두로 사용되는 매개변수
  - 함수 안에서 그 값이 사용도 되고 변경도 된다.
  - 입출력 매개변수도 포인터로 전달한다.

배열의 전달 (1/2)

- 함수의 매개변수는 배열 원소에 대한 포인터형으로 선언한다.

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

- 함수를 정의할 때 배열의 크기가 필요하면 배열의 크기도 매개변수로 받아와야 한다.

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

- 배열이 입력 매개변수일 때는 const 키워드를 지정한다.

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

배열의 전달 (2/2)

- 배열을 매개변수로 가진 함수를 호출할 때는 배열의 이름을 인자로 전달한다.

```
int x[5] = {1, 2, 3, 4, 5};
print_array(x, 5);
```

- 함수를 정의할 때는 매개변수인 포인터를 배열 이름인 것처럼 인덱스와 함께 사용한다.

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

예제 8-15 : 배열을 입력 매개변수로 사용하는 함수 (1/2)

```
02  #define SIZE 10
03  void copy_array(const int source[], int target[], int size);
04  void print_array(const int arr[], int size);
05
06  int main(void)
07  {
08      int x[SIZE] = { 10, 20, 30, 40, 50 };
09      int y[SIZE] = { 0 };
10
11      printf("x = ");
12      print_array(x, SIZE);
13      copy_array(x, y, SIZE);
14      printf("y = ");
15      print_array(y, SIZE);
16      return 0;
17  }
```

예제 8-15 : 배열을 입력 매개변수로 사용하는 함수 (2/2)

```
19  void copy_array(const int source[], int target[], int size)
20  {
21      int i;
22      for (i = 0; i < size; i++)
23          target[i] = source[i];
24  }
25
26  void print_array(const int arr[], int size)  // arr는 입력 매개변수
27  {
28      int  i;
29      for (i = 0; i < size; i++)
30          printf("%d ", arr[i]);
31      printf("\n");
32  }
```

실행결과

```
x = 10 20 30 40 50 0 0 0 0 0
y = 10 20 30 40 50 0 0 0 0 0
```