8장 포인터

박 종 혁 교수
서울과학기술대학교 컴퓨터공학과
UCS Lab
Tel: 970-6702
Email: jhpark1@seoultechackr

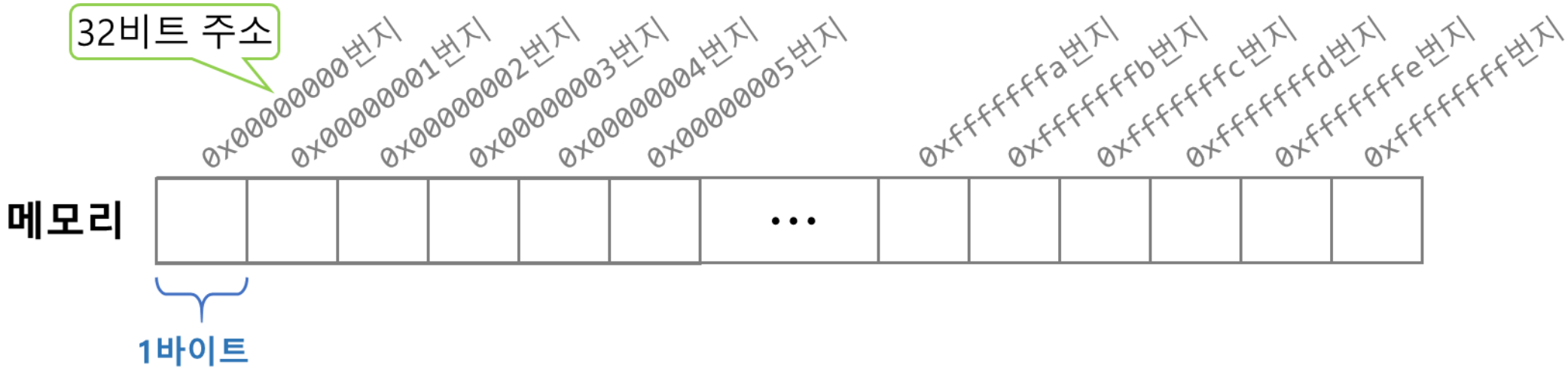
목차

포인터의 개념 (1/2)

❖포인터(pointer)는 주소(address)를 저장하는 변수

❖메모리에는 각각의 바이트를 구분하기 위한 주소(번지)가 있음

- 주소의 크기도 플랫폼에 따라 다름
- 32비트 플랫폼에서는 주소가 4바이트, 64비트 플랫폼에서는 주소가 8바이트

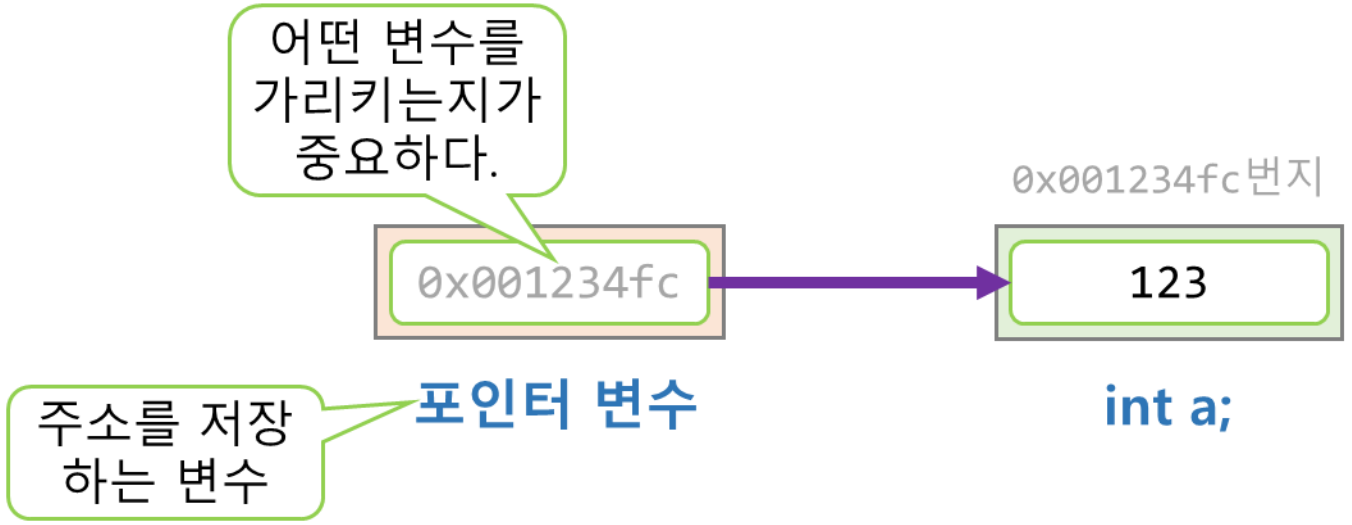


포인터의 개념 (2/2)

❖포인터를 사용할 때는 주소 값이 아니라 포인터가 어떤 변수를 가리키는지가 중요

❖포인터는 다른 변수를 가리키는 변수

- 포인터는 주소를 이용해서 다른 변수에 접근

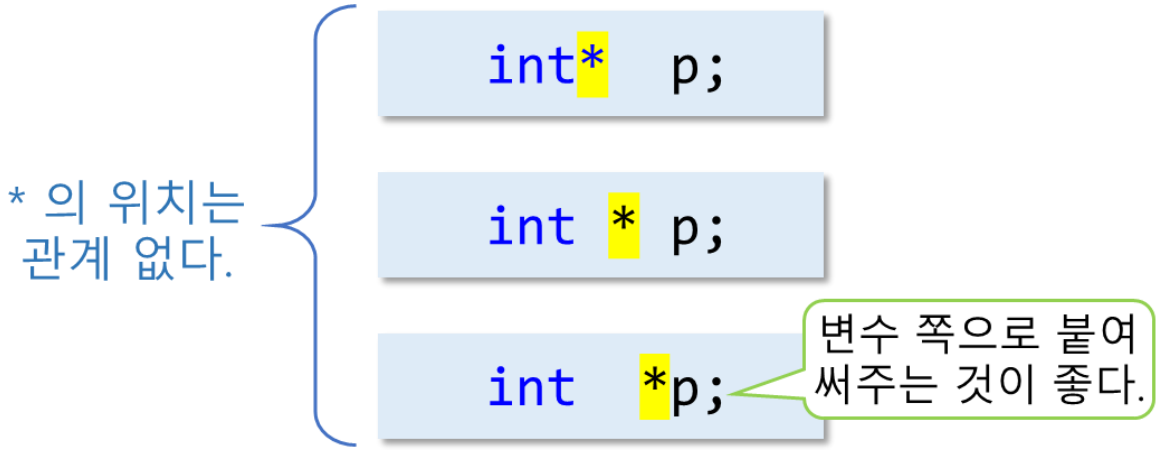


포인터의 선언

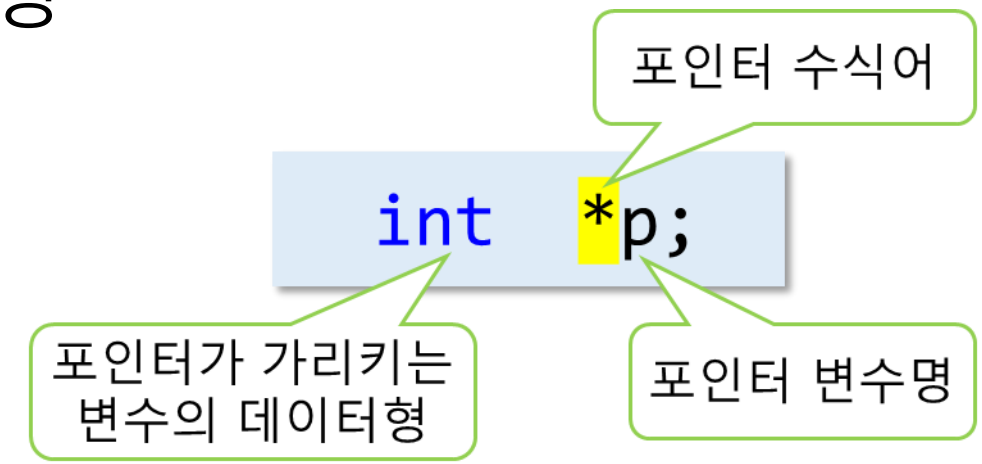
형식 데이터형 *변수명;
데이터형 *변수명 = 초기값;

사용예

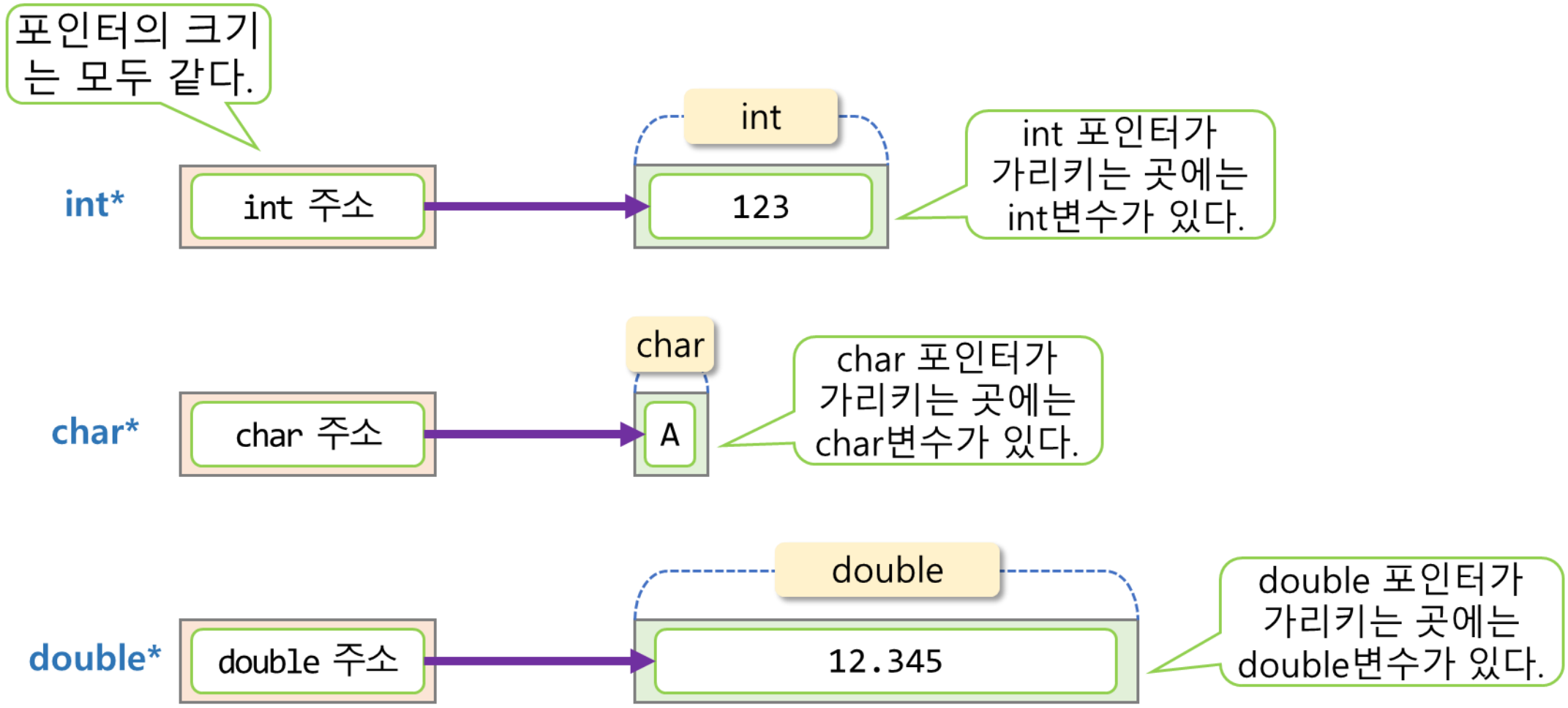
```
int *p;
double *pd;
int a = 123;
int *pa = &a;
char *pc = NULL;
```



- 포인터를 선언할 때 지정하는 데이터형은 포인터가 가리키는 변수의 데이터형



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 같음

- 포인터의 크기는 플랫폼에 의해서 결정
  - 32비트 → 4B, 64비트 → 8B

```
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 : 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*));          // 4바이트
14      printf("sizeof(double*) = %zd\n", sizeof(double*));    // 4바이트
15      printf("sizeof(char*) = %zd\n", sizeof(char*));        // 4바이트
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

- 포인터에 직접 절대 주소를 대입해서는 안됨

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

- 변수의 주소를 구할 때는 주소 구하기 연산자인 &를 이용

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

- 어떤 변수의 주소로 초기화할지 알 수 없으면 0으로 초기화

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

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

예제 : 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

❖주소 구하기 연산자 &

- 변수(l-value)에만 사용 가능

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

- 상수나 수식에는 사용 불가

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

❖역참조(간접 참조) 연산자 *

- 포인터가 가리키는 변수에 접근

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

p가 가리키는 int 변수를 출력

p가 가리키는 int 변수에 대입

- 포인터에만 사용할 수 있으며, 포인터가 아닌 변수나 수식에는 사용할 수 없음

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

x는 포인터가 아니므로 *를 사용할 수 없음

수식에는 *를 사용할 수 없음

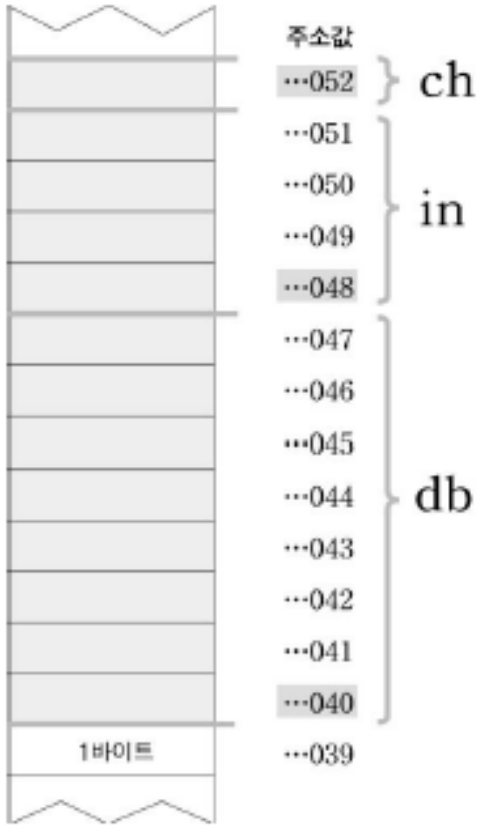
주소연산자&

- 특정 변수의 포인터를 구하기 위해서는 주소연산자(&)를 사용

- 포인터를 구하여 출력

```
char ch;
int in;
double db;
printf("ch의 포인터: %u\n", &ch);
printf("in의 포인터: %u\n", &in);
printf("db의 포인터: %u\n", &db);
```

```
ch의 포인터 : 1245052   // char형 변수의 주소값
in의 포인터 : 1245048   // int형 변수의 시작 주소값
db의 포인터 : 1245040   // double형 변수의 시작 주소값
```



역참조 연산의 의미

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

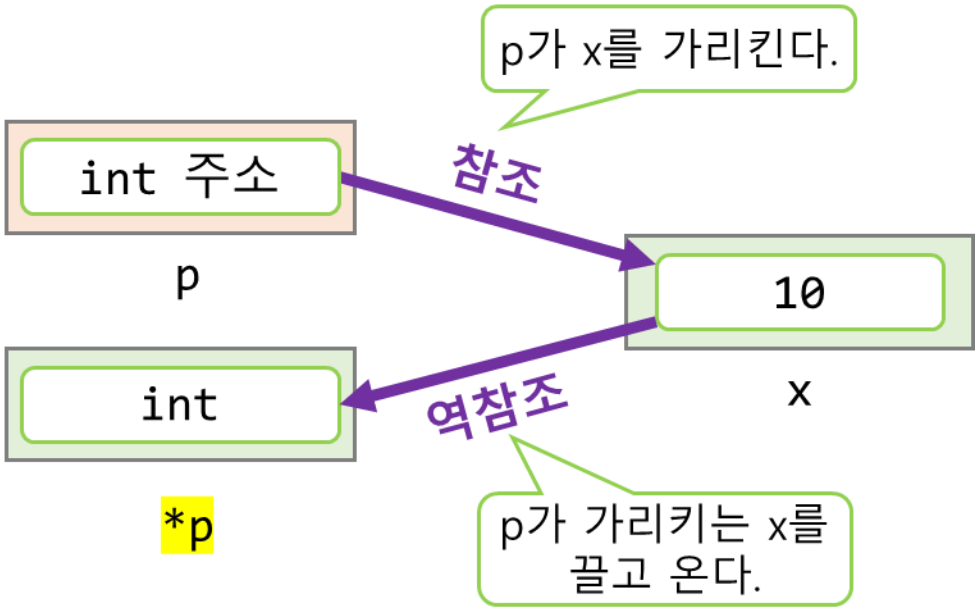
p가 x를 가리킨다.


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

x

x

*p가 사용되는 곳에 x가 대신 사용되는 것처럼 처리된다.



예제 : 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  }
```

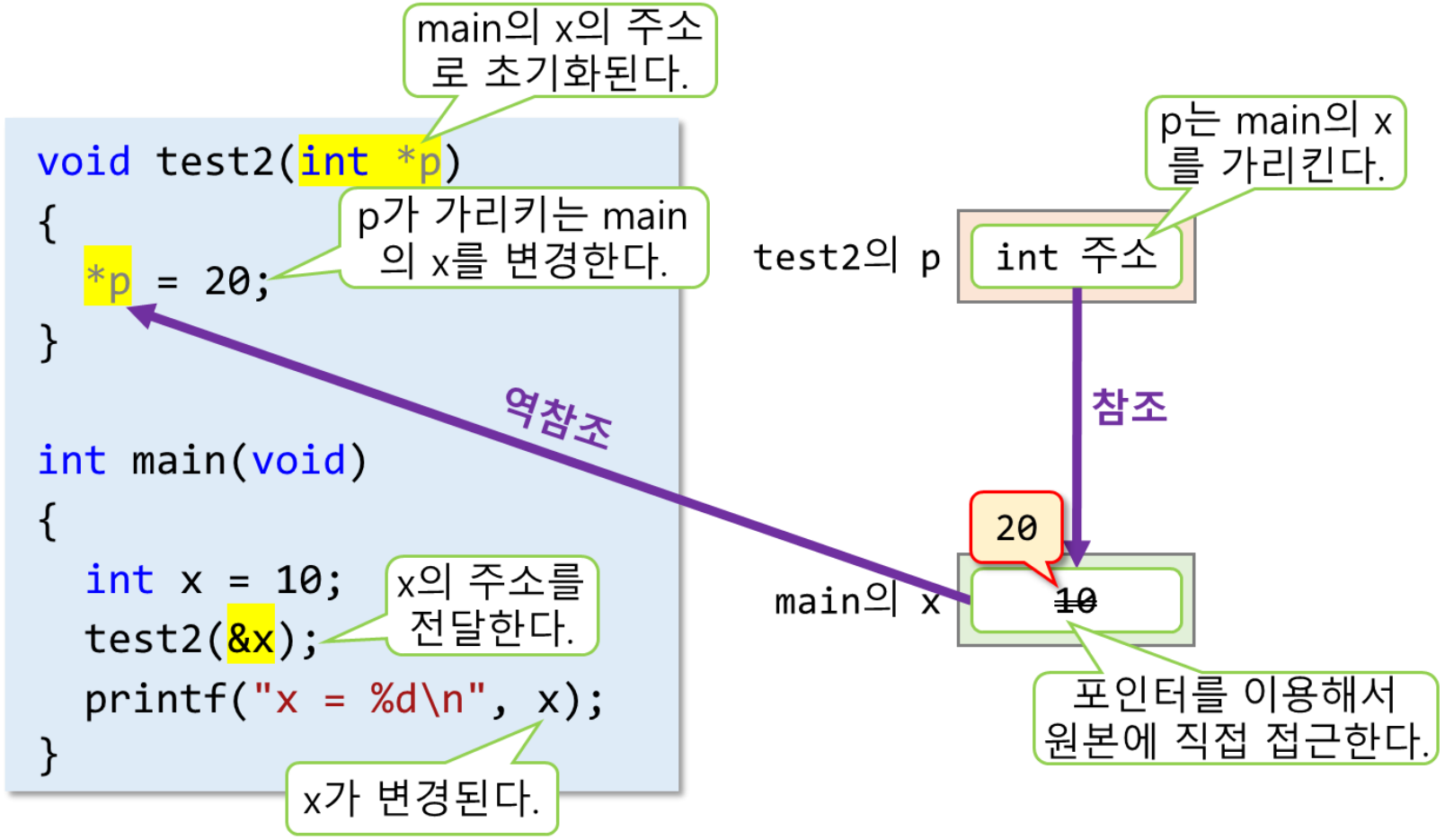
실행결과

```
 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 있음



예제 : 포인터가 필요한 경우

```
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  }
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
```

포인터 사용 시 주의 사항 (1/2)

❖포인터는 초기화하고 사용하는 것이 안전

- 포인터를 초기화하지 않고 사용하면 실행 에러가 발생

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

- 어떤 변수를 가리킬지 알 수 없으면 널 포인터로 초기화

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

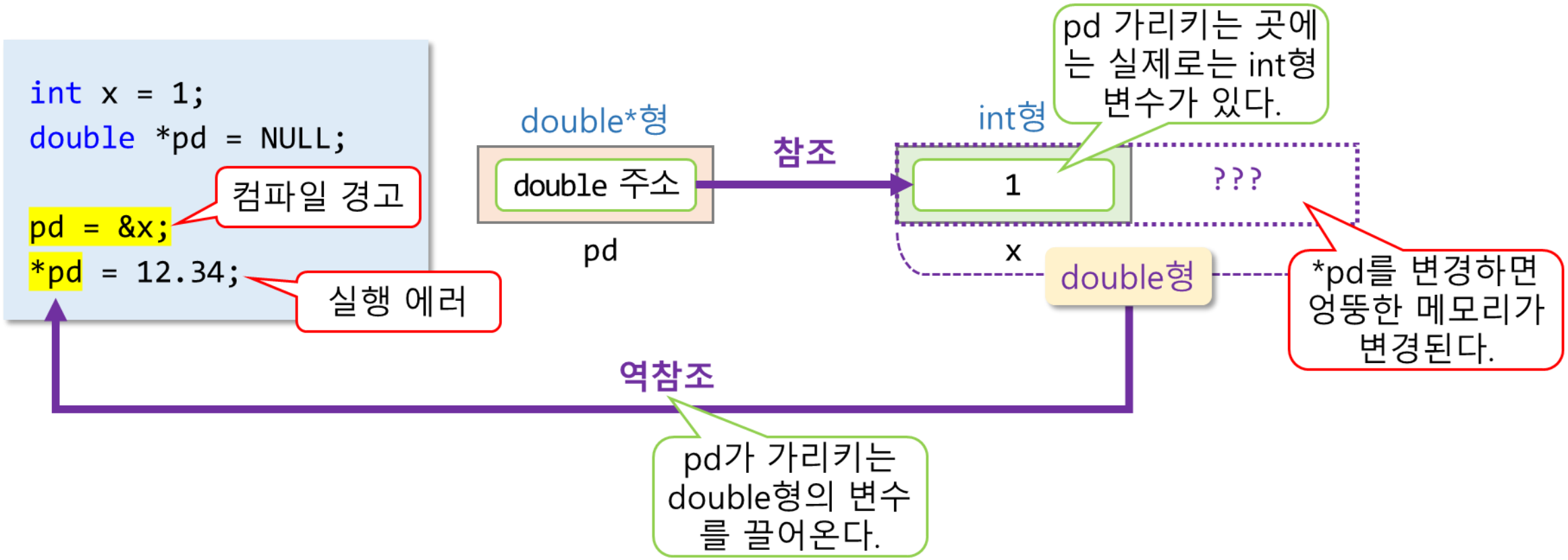
- 포인터를 안전하게 사용하려면 사용할 때 널 포인터인지 검사
  - 널 포인터가 아닌 경우에만 포인터가 가리키는 변수에 접근

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

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

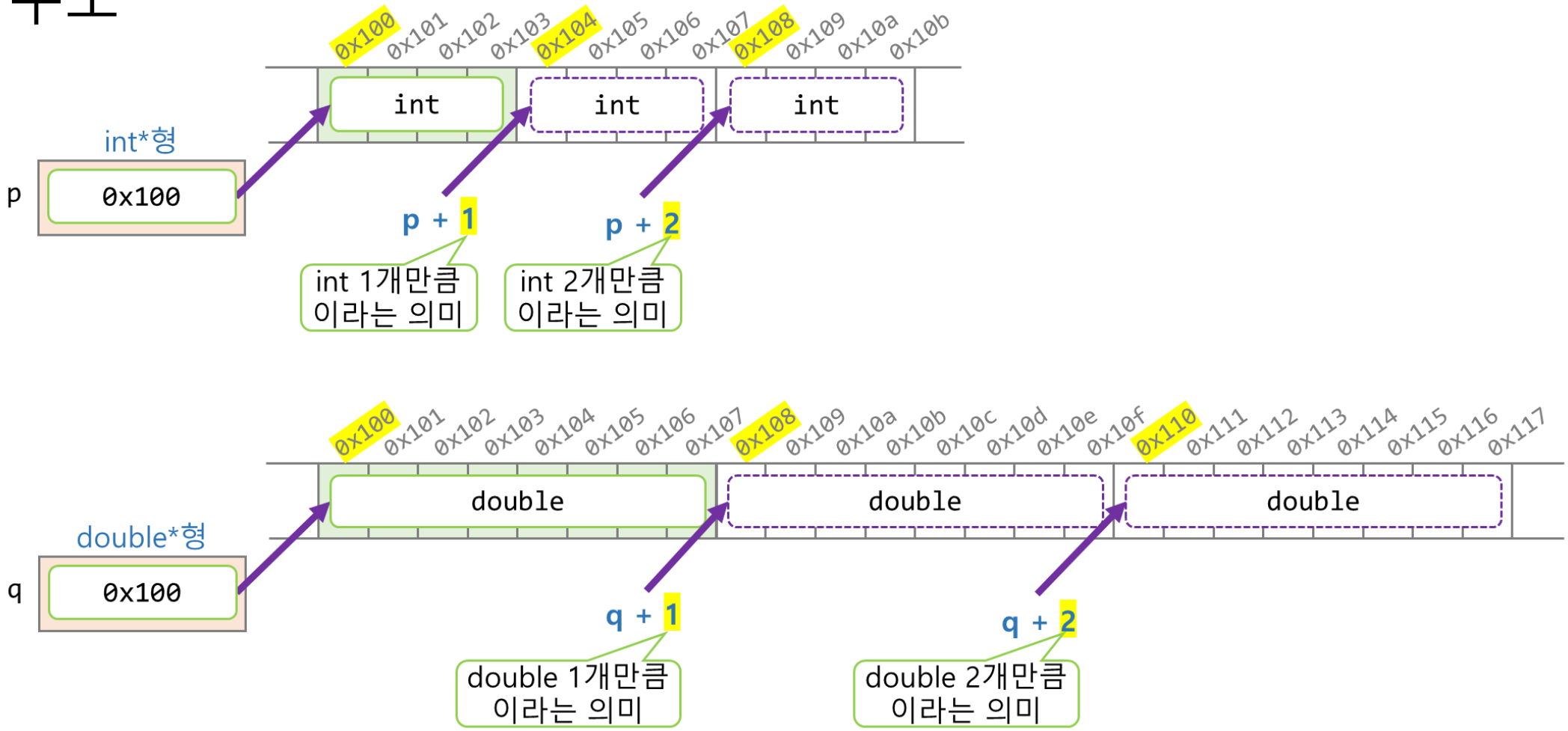
포인터 사용 시 주의 사항 (2/2)

- 포인터의 데이터형과 포인터가 가리키는 변수의 데이터형이 같아야 함



포인터와 +, – 연산 (1/2)

- p+N 연산의 결과는 p가 가리키는 데이터형 N개 크기만큼 더한 주소



예제 : '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

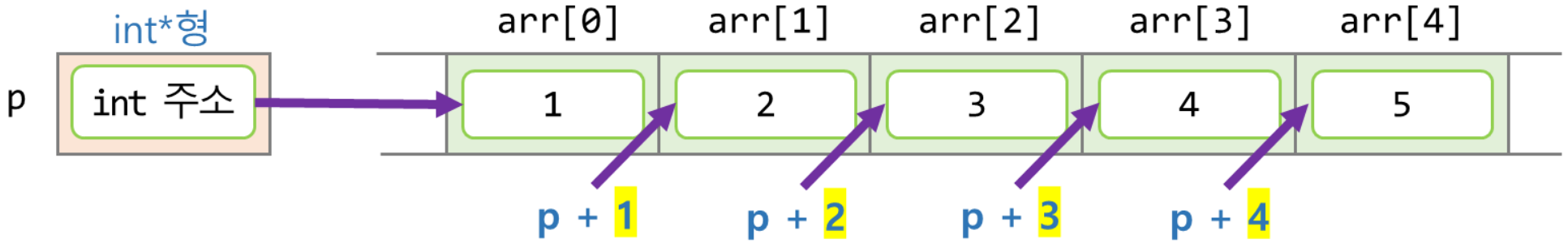
```
int*   : 00000100, 00000104, 00000108
double*: 00000100, 00000108, 00000110
char*  : 00000100, 00000101, 00000102
```

포인터와 +, – 연산 (2/2)

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

```
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 : 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);        // arr[i]의 주소를 출력
12          printf("*(p + %d) = %d\n", i, *(p + i));  // arr[i]를 출력
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
```

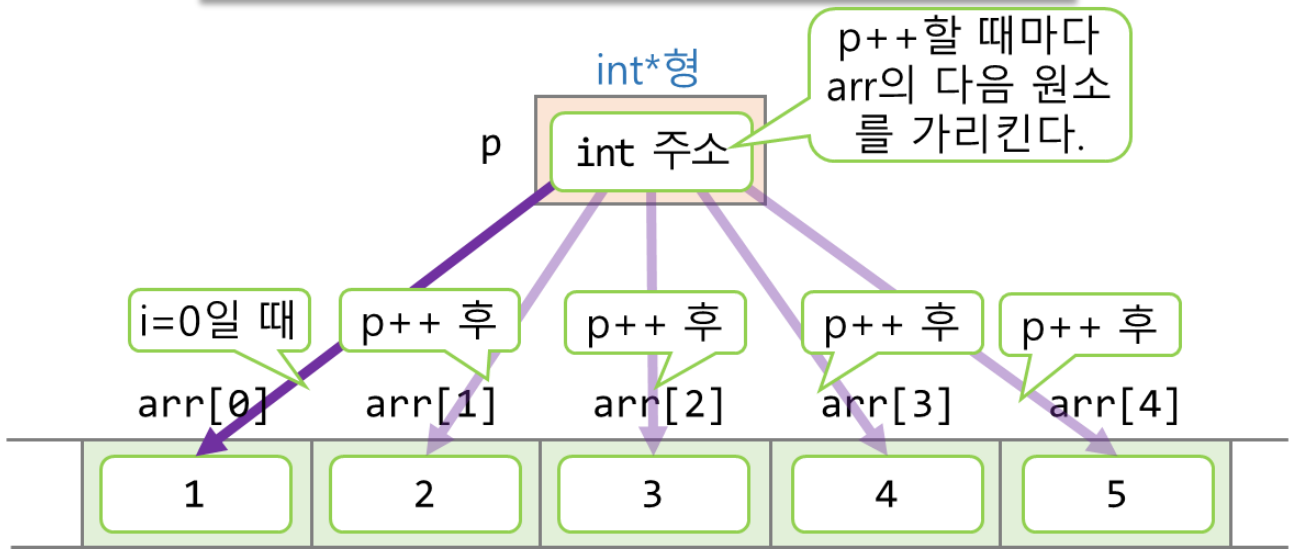
포인터와 ++, -- 연산 (1/2)

- p++, ++p
  - p가 가리키는 데이터형 1개 크기만큼 주소를 증가
- p--, --p
  - p가 가리키는 데이터형 1개 크기만큼 주소를 감소

```
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 : 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 (2/2)

❖증감 연산자와 역참조 연산자를 함께 사용할 때는 연산자 우선 순위를 신경 써야 함

- *p++은 *(p++)를 의미 // *(p+1) 포인터p의 주소값의 증가

```
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

- // 포인터p가 가리키는 곳의 저장된 값의 증가

배열처럼 사용되는 포인터 (1/3)

❖배열 원소를 가리키는 포인터

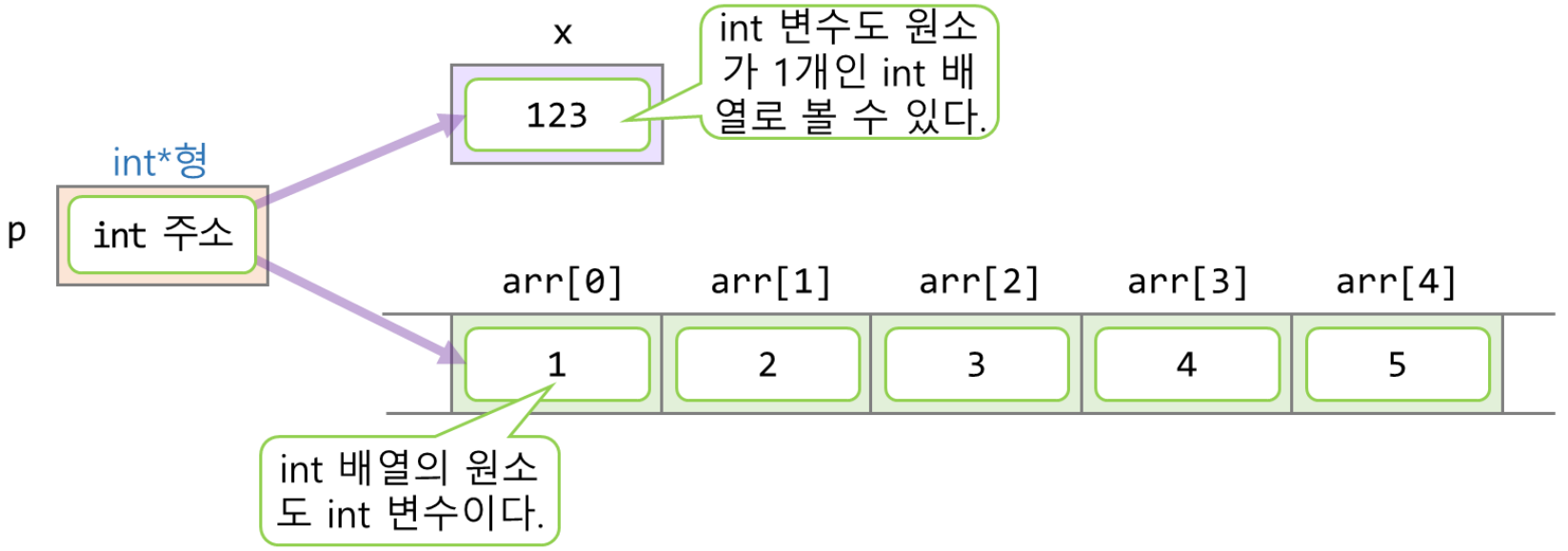
- type형의 포인터는 항상 type형의 변수 또는 type형 배열의 원소를 가리킬 수 있음

```
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 있음

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

p를 배열 이름인 것처럼 사용한다

p는 배열 원소를 가리키는 포인터

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

int 변수

p가 가리키는 배열의 i번째 원소

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

주소

p가 가리키는 배열의 i번째 원소의 주소

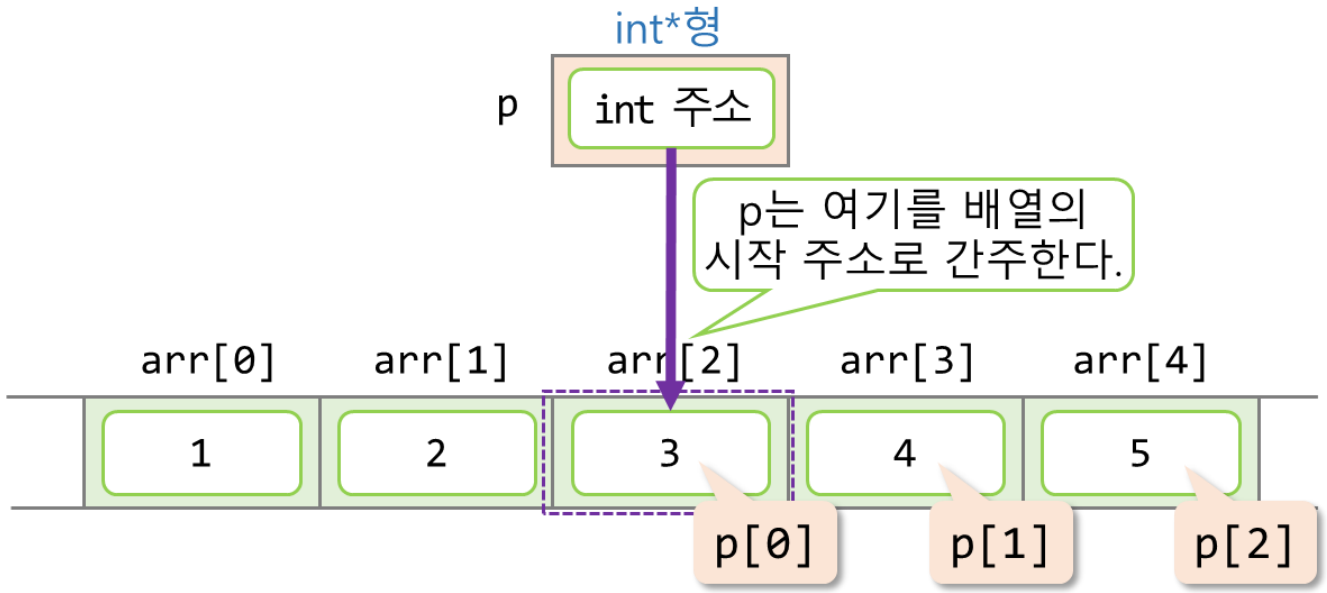
배열처럼 사용되는 포인터 (3/3)

- 배열의 원소를 가리키는 포인터는 배열의 어떤 원소든지 가리킬 수 있음

```
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 : 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
```

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

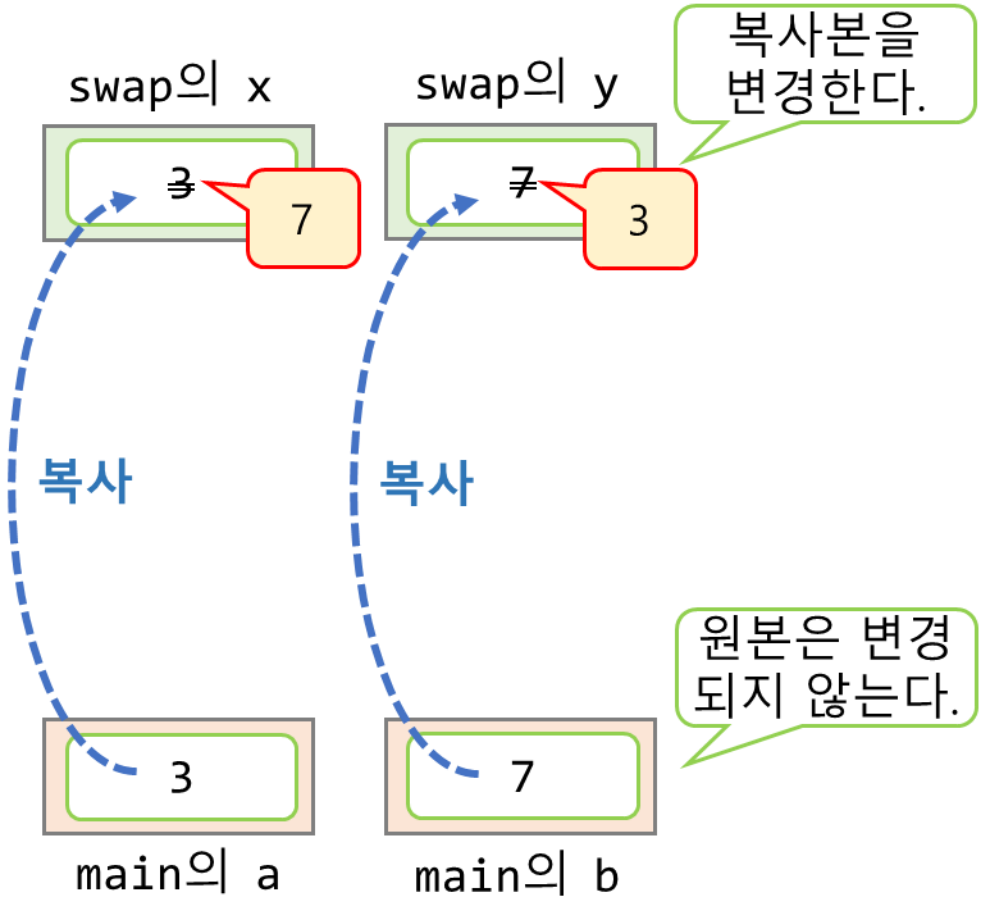
값에 의한 전달 : swap 함수

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;



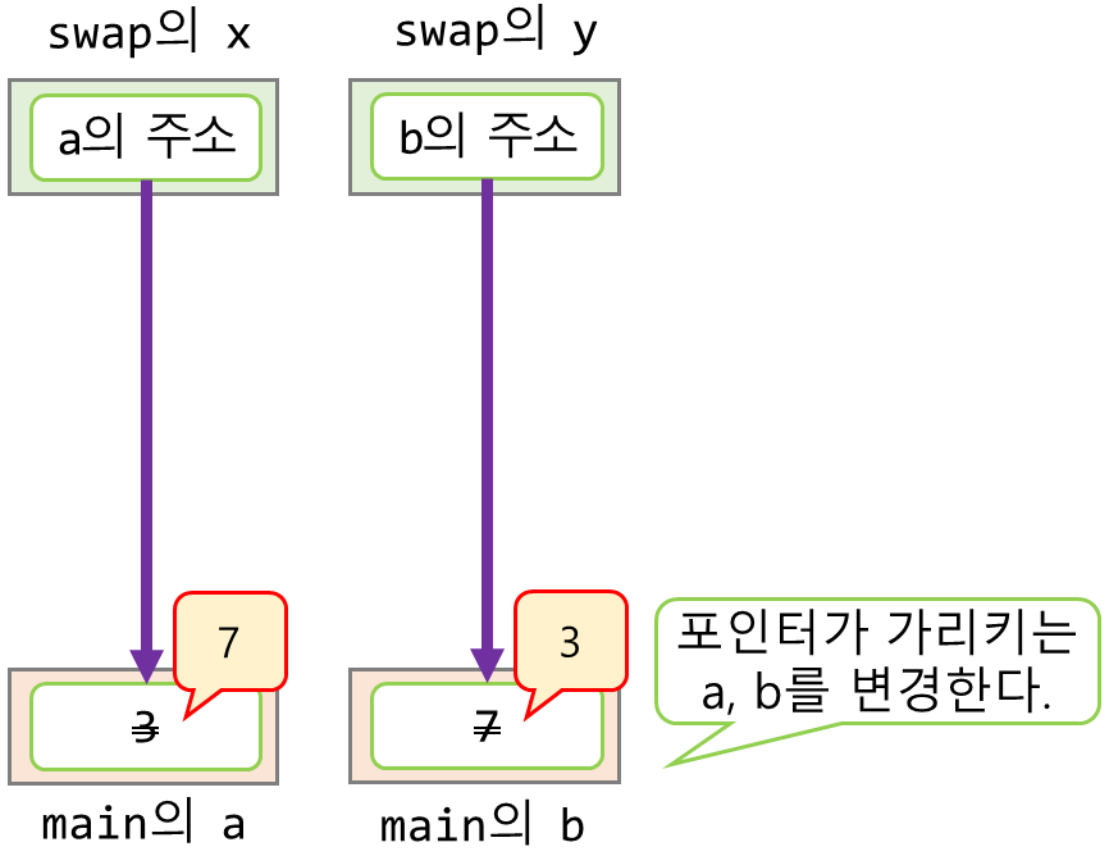
포인터에 의한 전달 : swap 함수

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 : 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

- 함수의 원형을 정할 때, 처리 결과를 저장할 변수를 가리키는 포인터형으로 매개변수를 선언
- 함수를 호출할 때, 처리 결과를 받아올 변수의 주소를 전달
- 함수를 정의할 때, 포인터형의 매개변수가 가리키는 곳에 처리 결과를 저장

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

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

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

예제 : 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
```

배열의 전달 (1/2)

- 함수의 매개변수는 배열 원소에 대한 포인터형으로 선언

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

- 함수를 정의할 때 배열의 크기가 필요하면 배열의 크기도 매개변수로 받아와야 함

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

배열의 전달 (2/2)

- 배열을 매개변수로 가진 함수를 호출할 때는 배열의 이름을 인자로 전달

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

- 함수를 정의할 때는 매개변수인 포인터를 배열 이름인 것처럼 인덱스와 함께 사용

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

배열 vs 포인터

- 배열 이름은 특정 변수 전용 포인터인 것처럼 사용할 수 있음
- 배열의 시작 주소는 변경할 수 없음

```
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 있음

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

p = y;  // OK
```

예제 : 배열과 포인터의 차이점

```
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
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
```

배열과 포인터의 관계

- a와 p는 포인터이고 둘 다 첨자를 붙일 수도 있음

```
a[i] ←→ *(a + i)
p[i[ ←→ *(p + i)
```

- 포인터 변수는 다른 값을 가질 수 있지만, 배열 이름은 안됨

```
p = a + i ;
a = q ;          /* error */
```

배열과 포인터의 관계

- 예제 코드 (배열의 합 구하기)

```
#define    N          100
int        * p, a[N], sum ;
```

- Version 1

```
for ( i=0, sum=0; I<N; ++i )
    Sum += a[i];        /* 또는 sum += *(a+i); */
```

- Version 2

```
for ( p=a, sum=0; p < &a[N]; ++p )
    Sum += *p;
```

- Version 1

```
for ( p=a, i=0, sum=0; i<N; ++I )
    Sum += p[i];
```

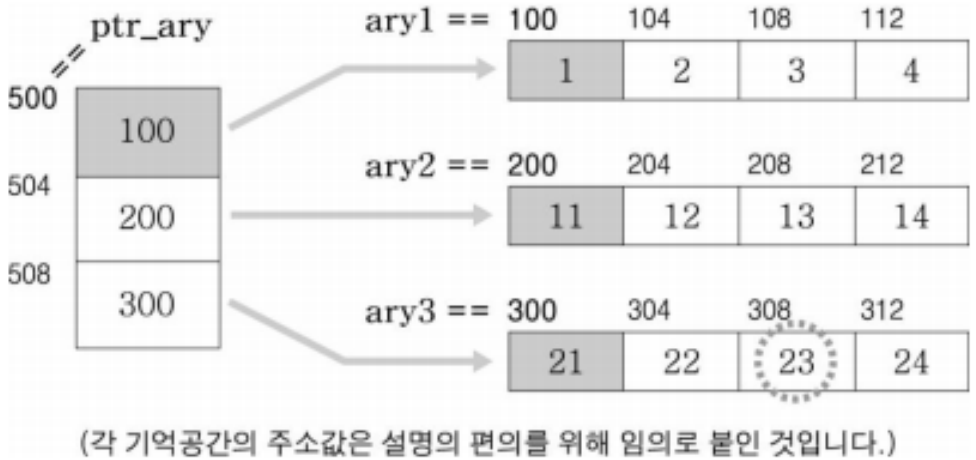
포인터 배열

- 배열의 원소의 형은 포인터형을 포함하여 임의의 형이 될 수 있음
- 포인터 배열은 문자열을 다룰 때 많이 사용됨

포인터 배열

❖ 1차원 배열의 배열명을 포인터배열에 저장하면 포인터배열을 2차원 배열처럼 사용할 수 있음

```
int ary1[4]={1,2,3,4};
int ary2[4]={11,12,13,14};
int ary3[4]={21,22,23,24};
int *ptr_ary[3]={ary1,ary2,ary3};
//각 배열명을 포인터배열에 초기화
```



(각 기억공간의 주소값은 설명의 편의를 위해 임의로 붙인 것입니다.)

❖ ary3배열의 세 번째 배열요소(23값)를 참조하는 과정

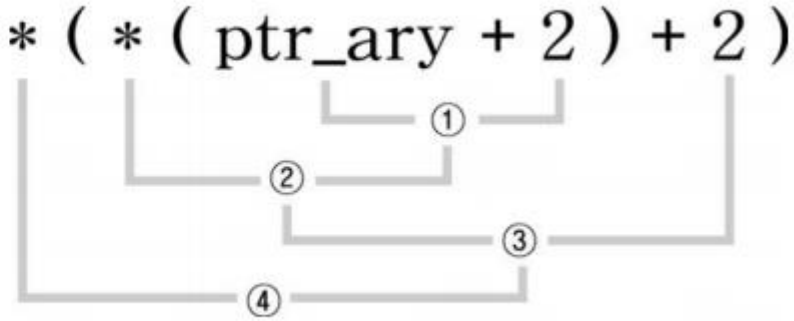
1. 먼저 ptr_ary 배열의 세 번 째 배열요소를 참조 ⇨ ptr_ary[2]
2. 참조된 배열요소 ptr_ary[2]는 배열명ary3를 저장한 포인트 변수이므로 배열명처럼 사용하여 ary3의 세 번째 배열요소를 참조

   ⇨ **printf("%d",ptr_ary[2][2]);**

포인터 배열

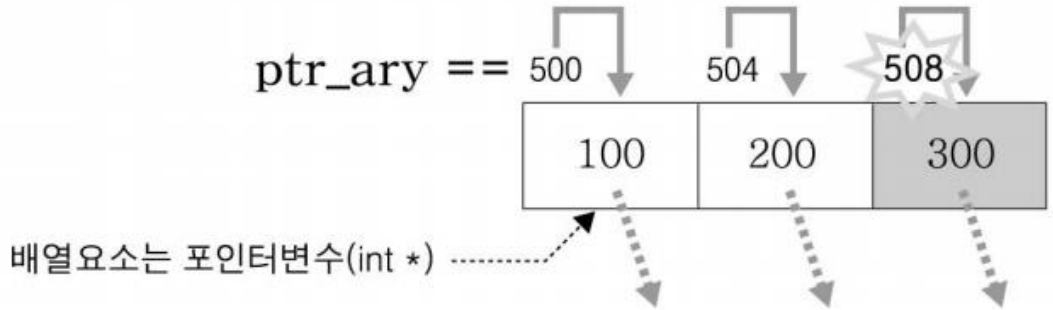
❖ptr_ary[2][2]가 참조되는 과정의 주소값을 계산

- **일단 포인터 표현으로 바꾸고 연산 순서를 따라감**



- **1번 연산 : 포인터배열의 세 번째 배열요소를 가리키는 포인터가 구해짐**

ptr_ary+2 = ptr_ary+(2*sizeof(ptr_ary[0])) = 500+(2*4) = 508

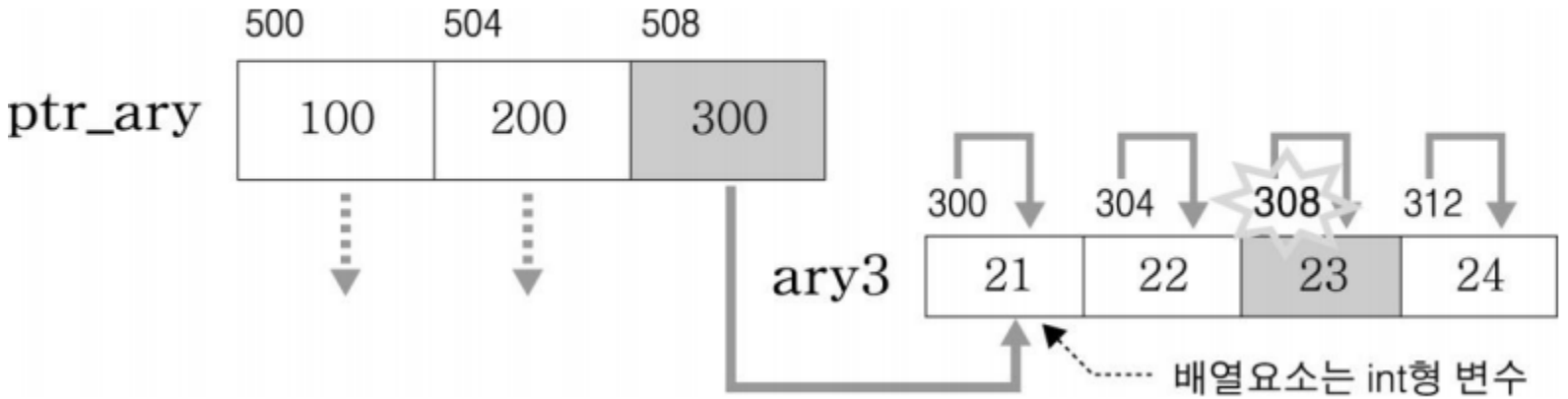


포인터 배열

- 2번 연산 : 포인터 배열의 세 번째 배열요소의 값 300번지가 구해짐

- 3번 연산 : ary3배열의 세 번째 기억공간을 가리키는 포인터가 구해짐

300+2 => 300+(2*sizeof(ary3[0])) => 308



- 4번 연산 : 308번지는 포인터이므로 참조연산을 수행하면 값23이 참조됨

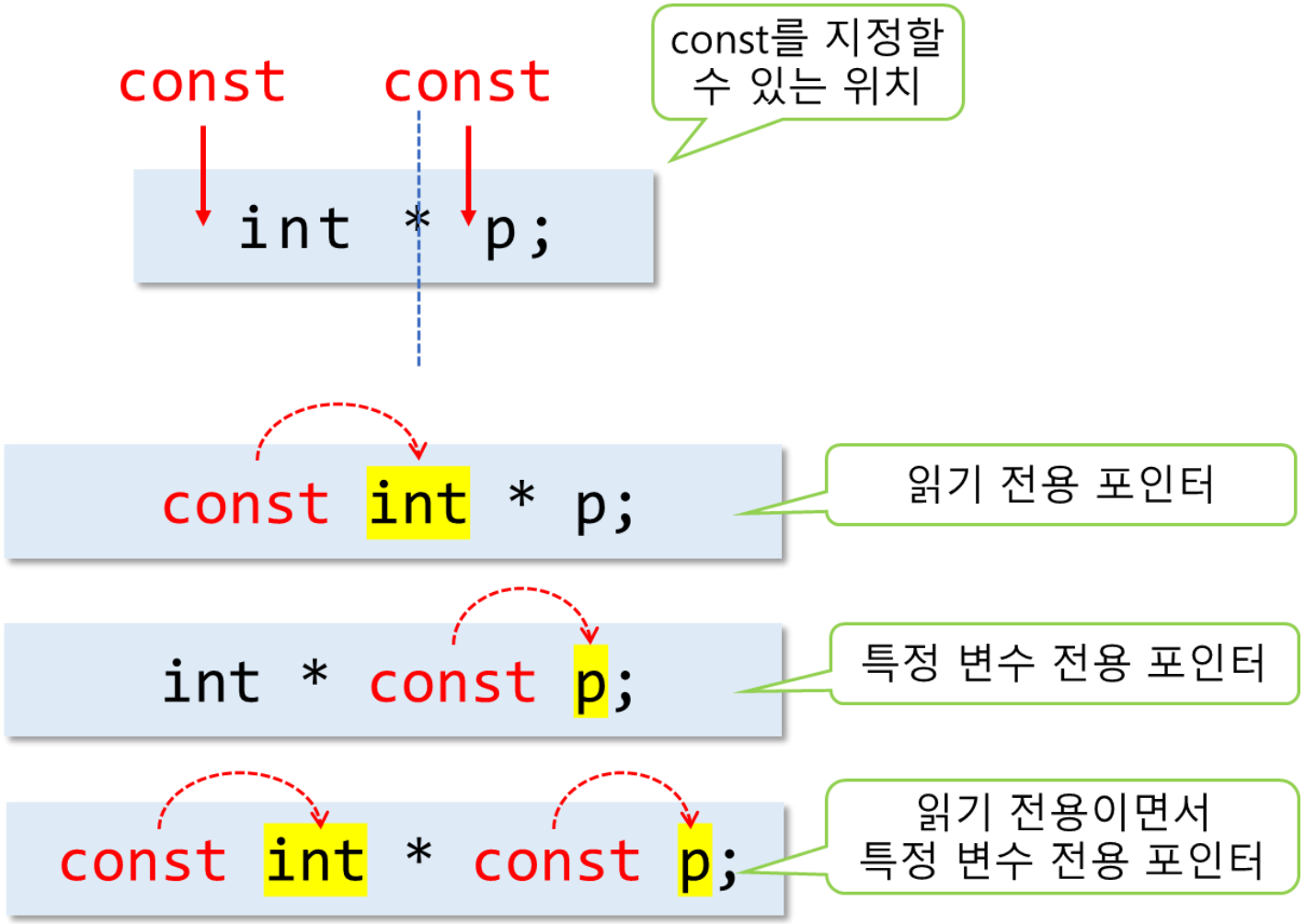
포인터 배열

```
#include <stdio.h>

int main()
{
        int ary1[4]={1,2,3,4};
        int ary2[4]={11,12,13,14};
        int ary3[4]={21,22,23,24};
        int *ptr_ary[3]={ary1,ary2,ary3};                    // 포인터 배열에 각 배열명을 초기화한다
        int i, j;                                            // 반복 제어 변수

        for(i=0;i<3;i++){
                for(j=0;j<4;j++){
                        printf("%5d",ptr_ary[i][j]);         // 3행 4열의 2차원 배열처럼 출력할 수 있다
                }
                printf("\n");    // 한 행을 출력한 후에 줄을 바꾼다
        }
        return 0;
}
```

출력 결과

```
 1  2  3  4
11 12 13 14
21 22 23 24
```

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;

❖읽기 전용 포인터

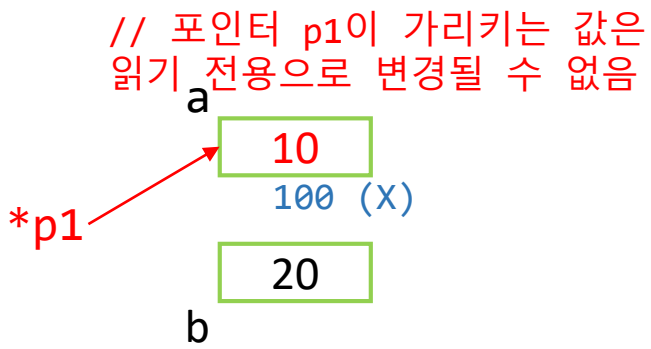
- 포인터가 가리키는 변수의 값을 변경할 수 없음

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

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

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

a를 직접 변경할 수는 있다

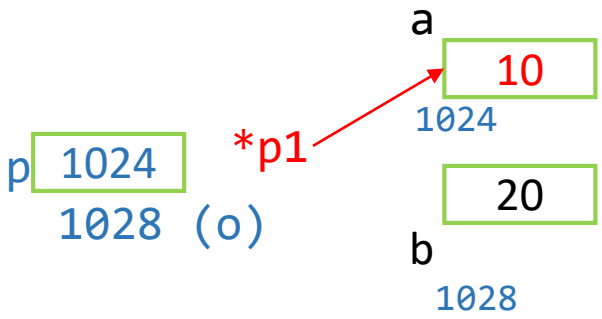
p1으로 접근할 때는 변경할 수 없다



const 데이터형 *변수;

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

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



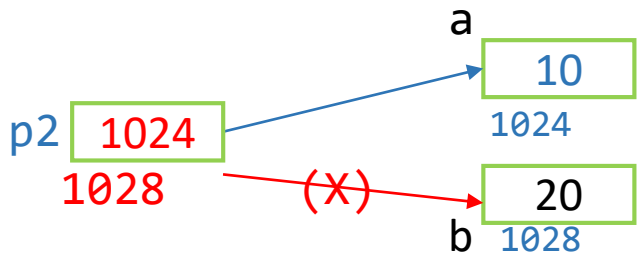
- 선언 시 널 포인터로 초기화하고, 원하는 시점에 특정 변수의 주소를 저장하고 사용할 수 있음

데이터형 * const 변수;

❖특정 변수의 전용 포인터

- 포인터 자신의 값(포인터에 저장된 주소)을 변경할 수 없음
- 다른 변수를 가리킬 수 없음

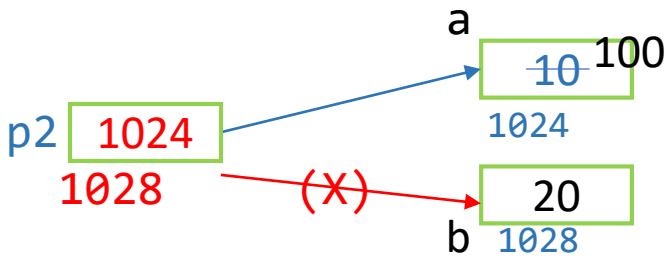
```
int *const p2 = &a;    // a 전용 포인터
p2 = &b;               // 컴파일 에러
```



데이터형 * const 변수;

- 포인터가 가리키는 변수의 값은 변경할 수 있음

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



- 선언 시 반드시 참조하려는 값으로 초기화해야 함

```
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 없음

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

예제 : const 포인터의 의미

```
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);
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  }
```

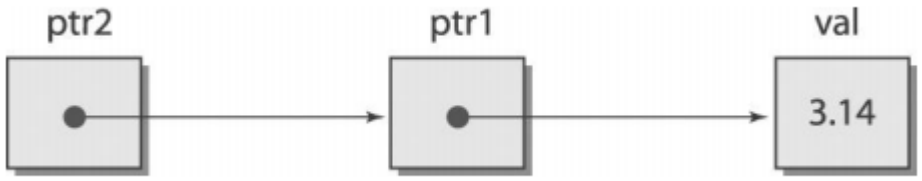
실행결과

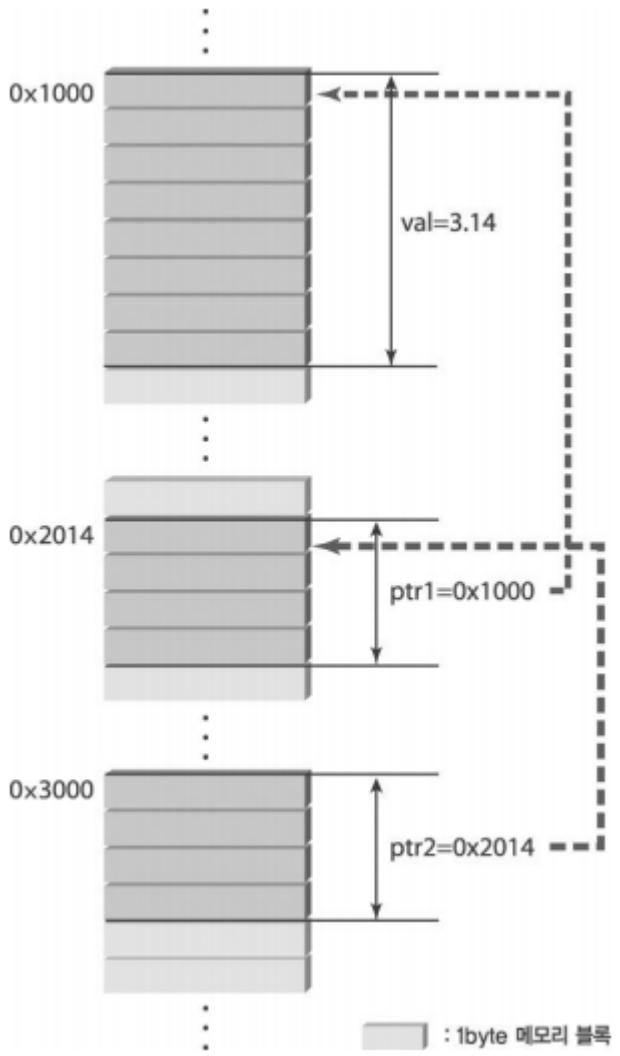
```
*p1 = 10
*p1 = 20
*p2 = 100
*p3 = 100
```

이중 포인터

- 포인터의 포인터
- 더블 포인터
- 싱글 포인터의 주소 값을 저장하는 용도의 포인터

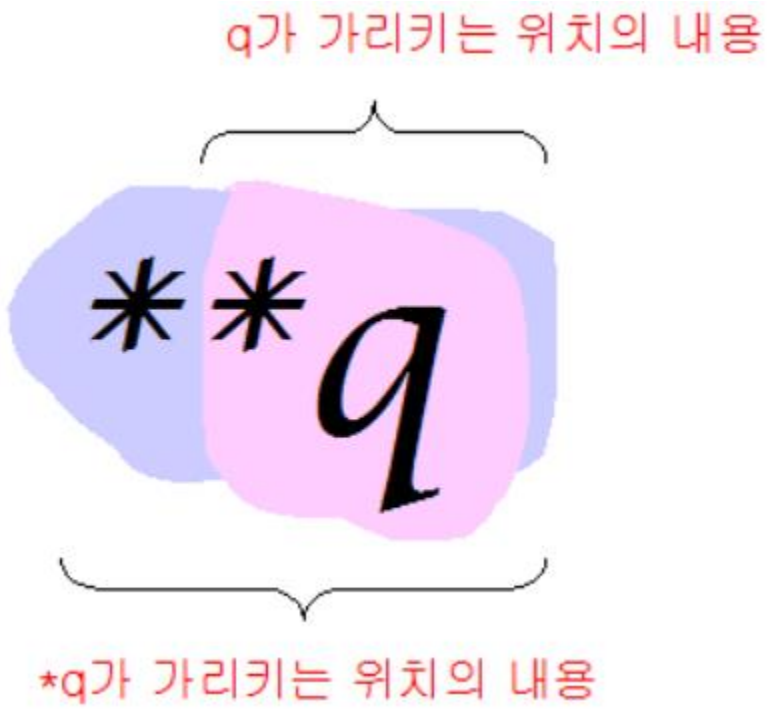
```
int main(void)
{
    double val=314;
    double *ptr1=&val       // 싱글 포인터
    double **ptr2=&ptr1;  // 더블 포인터
}
```





이중 포인터

- 이중 포인터의 해석



이중 포인터 – 사용 예제

```
#include <stdio.h>

int main(void)
{
        int i=100;
        int *p=&i;
        int **q=&p;

        *p=200;
        printf("i=%d  *p=%d  **q=%d \n", i, *p, **q);

        **q=300;
        printf("i=%d  *p=%d  **q=%d \n", i, *p, **q);

        return 0;
}
```

출력 결과

```
i=200  *p=200  **q=200
i=300  *p=300  **q=300
```

main() 함수의 인자

❖main()은 운영체제와의 통신을 위해 argc와 argv라는 인자를 사용함

❖예제 코드

```
void main(int argc, char *argv[]) {
    int i;
    printf("argc = %d\n", argc);
    for (i = 0; i < argc; ++i)
       printf("argv[%d] = %s\n", i, argv[i]);
}
```

- argc : 명령어 라인 인자의 개수를 가짐
- argv : 명령어 라인을 구성하는 문자열들을 가짐

main() 함수의 인자

- 앞의 프로그램을 컴파일하여 my_echo로 한 후, 다음 명령으로 실행:

  $ my_echo a is for apple

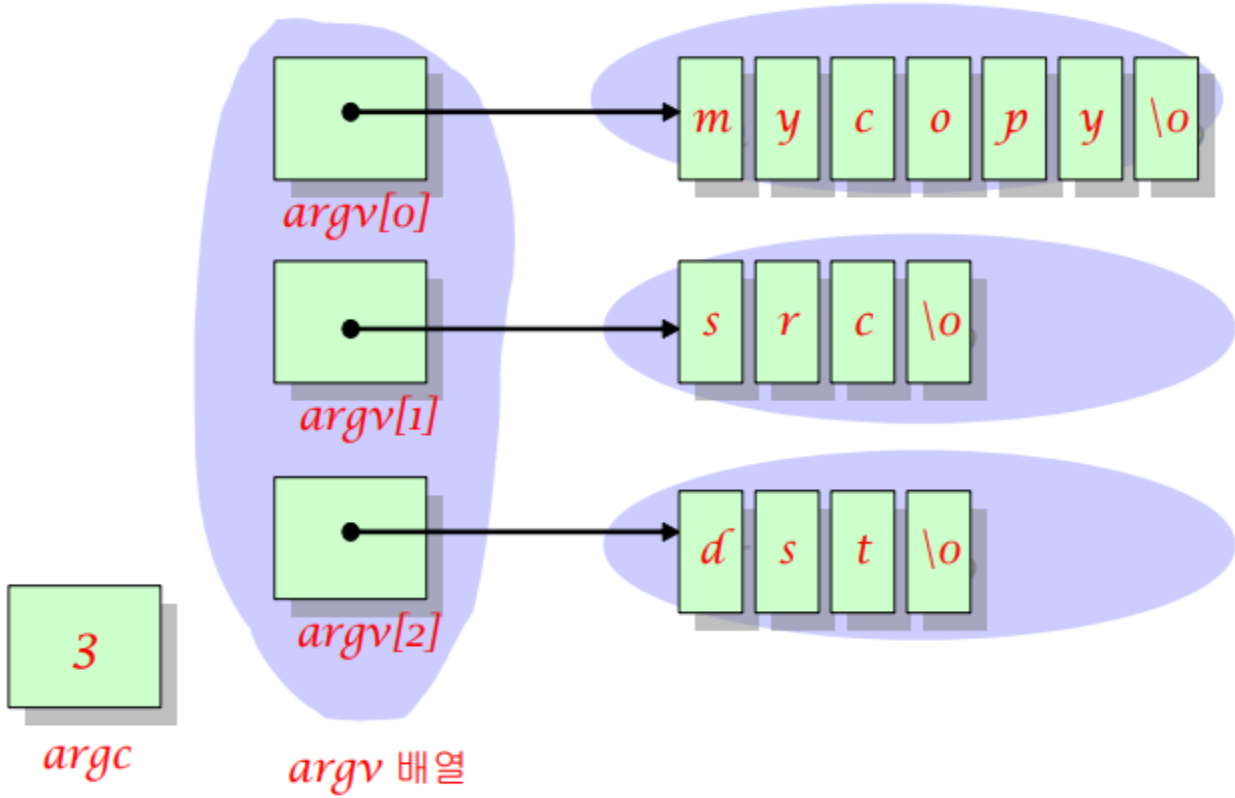
- 출력:

```
argc = 5
argv[0] = my_echo
argv[1] = a
argv[2] = is
argv[3] = for
argv[4] = apple
```

인수 전달 방법

- 다른 예제

```
C: \cprogram> mycopy src dst
```



main_argc

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        int i = 0;
        for(i = 0;i < argc; i++)
                printf("명령어 라인에서 %d번째 문자열 = %s\n", i, argv[i]);
        return 0;
}
```

```
c:\cprogram\mainarg\Debug>mainarg src dst
명령어 라인에서 0번째 문자열 = mainarg
명령어 라인에서 1번째 문자열 = src
명령어 라인에서 2번째 문자열 = dst
c:\cprogram\mainarg\Debug>
```

calloc()과 malloc()

- **stdlib.h에 정의되어 있음**
  - calloc : contiguous allocation
  - malloc : memory allocation
- **프로그래머는 calloc()과 malloc()을 사용하여 배열, 구조체, 공용체를 위한 공간을 동적으로 생성함**

calloc()과 malloc()

❖ **각 원소의 크기가 el_size인 n개의 원소를 할당 하는 방법**

Calloc(n, el_size);

Malloc(n *el_size);

- calloc()은 모든 원소를 0으로 초기화 하는 반면 malloc()은 하지 않음

❖ **할당받은 것을 반환하기 위해서는 free()를 사용**

예제 – calloc()과 malloc()

```
#include <stdio.h>
#include <stdlib.h>
Int main(void){
        int *a;                                 /* to be used as an array */
        int n ;                                 /* the size of the array */
        …                                 /* get n from somewhere, perhaps
                                                  interactively from the user */
        a = calloc(n, sizeof(int));                 /* get space for a */

        free(a);

}
```

calloc()과 malloc()

❖ Calloc함수는 배열을 할당 받고 초기화

```
void *calloc(unsigned int, unsigned int);
```

- 첫 번째 배열요소의 개수, 두 번째는 배열요소의 크기를 전달인자로 줌
- double형 변수 5개로 사용할 배열을 할당 받는 경우

```
double *dp;
dp=(double *)calloc(5, sizeof(double));
```

배열요소의 개수

double형 변수 하나의 크기

```
double *ap;
int i;
ap=(double *)calloc(5, sizeof(double))
for(i=0;i<5;i++){
        printf("%lf\n",ap[i]);
}
```

출력 결과

```
0.000000
0.000000
0.000000
0.000000
0.000000
```

모두 0으로 초기화 된다

calloc()과 malloc()

```
#include <stdio.h>
#include <stdlib.h>                              // malloc 함수를 사용하기 위해서 포함시킨다

int main()
{
        int *ip;                                 //int 형을 가리킬 포인터 변수
        double *dp;                              //double 형을 가리킬 포인터 변수

        ip=(int *)malloc(sizeof(int));           //기억공간을 동적으로 할당 받아서
        dp=(double *)malloc(sizeof(double));     //각 포인터 변수에 연결시킨다

        *ip=10;                  //포인터 변수로 각각 할당 받은 기억공간을
        *dp=34;                  //참조하여 값을 저장한다

        printf("정수형으로사용: %d\n", *ip);
        printf("실수형으로사용: %lf\n", *dp);
        return 0;
}
```

출력 결과

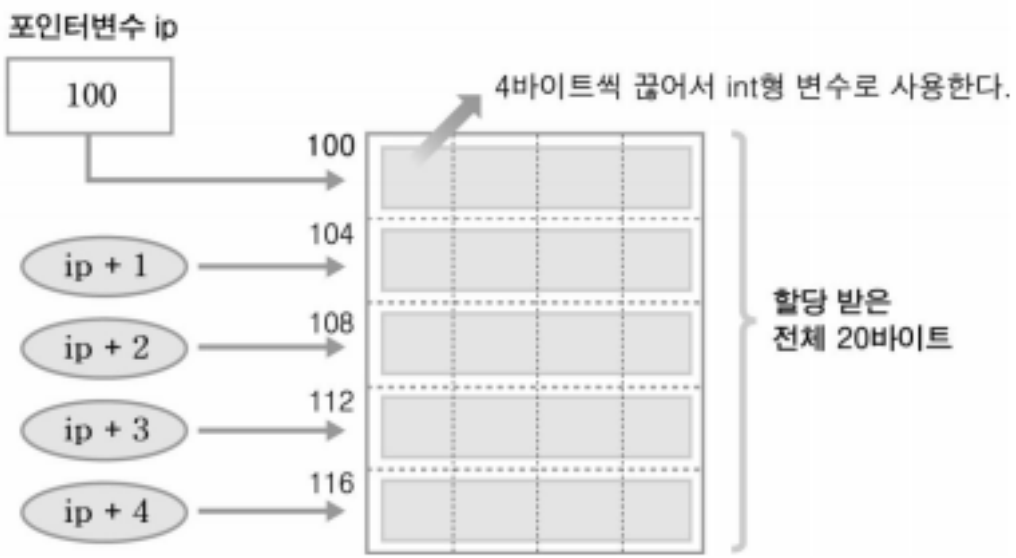
```
정수형으로 사용 : 10
실수형으로 사용 :
3.400000
```

calloc()과 malloc()

❖ 메모리의 동적할당은 많은 기억공간을 한꺼번에 할당 받아서 배열로 사용하는 것이 효율적

- int 형 변수 5개를 동적으로 할당 받는 경우

```
int *ip;                          //포인터 변수 선언
ip = (int *)malloc(20);           //20바이트를 한꺼번에 할당 받음
```



calloc()과 malloc()

```
#include <stdio.h>
#include <stdlib.h>                                  // malloc 함수를 사용하기 위해서 포함시킨다
int main(){
        int *ip;
        int i, sum=0;
        ip=(int*)malloc(5*sizeof(int));              // 전체 20바이트의 기억공간 할당
        if(ip==0){                                   // 메모리가 할당 되었는지 확인하여
                printf("메모리가부족합니다!\n");      // 메모리가 부족하면 메시지를 출력하고
                return 1;                            // 프로그램을 종료한다
        }
        printf("다섯명의나이를입력하세요: ");
        for(i=0; i<5; i++){
                scanf("%d", ip+i);                   // 데이터를 저장할 포인터를 전달한다
                sum+=ip[i];                                  // 입력된 값을 참조하여 누적한다
        }
        printf("다섯명의평균나이: %d\n", sum/5);      // 평균나이 출력
        free(ip);                                    // 할당 받은 메모리 반환
        return 0;
}
```

출력 결과

```
다섯 명의 나이를 입력하세요 : 21 27 24 22 35 (엔터)
다섯 명의 평균나이 : 25.8
```

추가 자료

calloc()과 malloc()

❖ 메모리 동적 할당을 사용하면 입력되는 문자열의 길이에 맞게 기억공간을 할당할 수 있음

1 문자열을 입력 받기 전에는 그 길이를 알 수 없으므로 우선 충분한 크기의 문자배열이 필요

2 문자배열에 문자열을 입력

3 문자열의 길이를 계산하여 그 크기에 맞는 기억공간을 동적으로 할당

4 동적으로 할당 받은 기억공간에 입력 받은 문자열을 복사

포인터의 용도

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

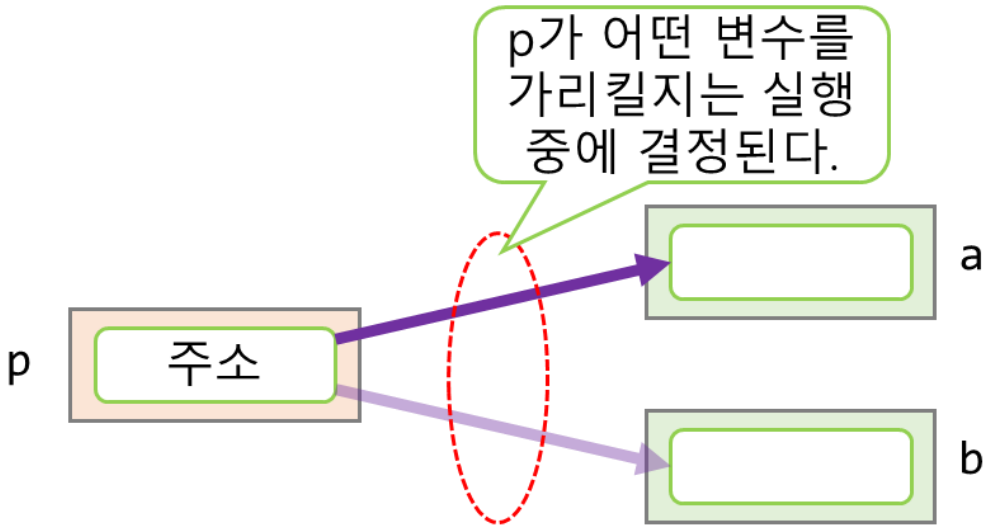
```
if (조건식)
   p = &a;
else
   p = &b;

do_something(*p);
result = *p * rate + extra;
*p = some_value;
```

조건식에 따라 p는 a 또는 b를 가리킨다.

p가 가리키는 변수로 함수를 호출한다.

p가 가리키는 변수로 수식을 계산한다.

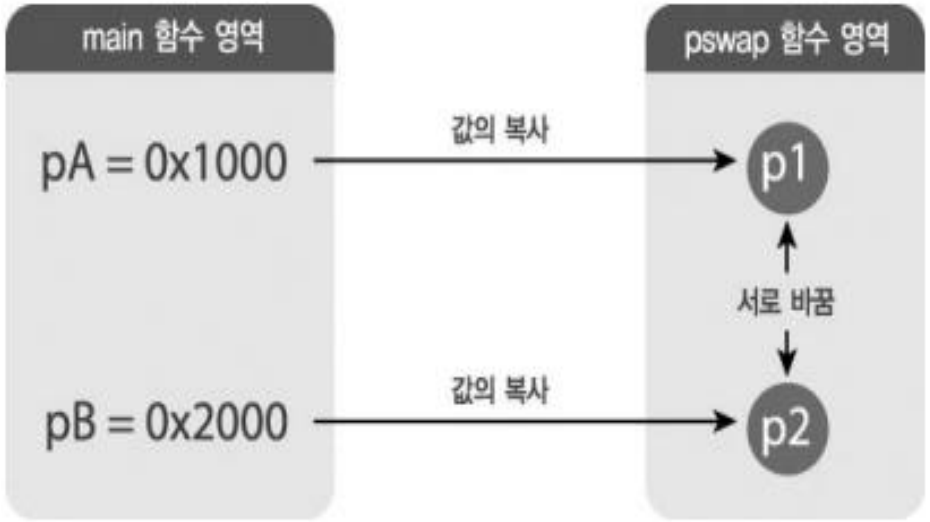
p가 가리키는 변수에 값을 대입한다.



구현 사례 1 : 효과 없는 swap 함수의 호출

```
/* ptr_swap1c */
#include <stdio.h>
void pswap(int *p1, int *p2);
int main(void)
{
    int A=10, B=20;
    int *pA, *pB;
    pA=&A, pB=&B;

    pswap(pA, pB);

    printf("pA가 가리키는 변수 : %d \n", *pA);
    printf("pB가 가리키는 변수 : %d \n", *pB);
    return 0;
}
```

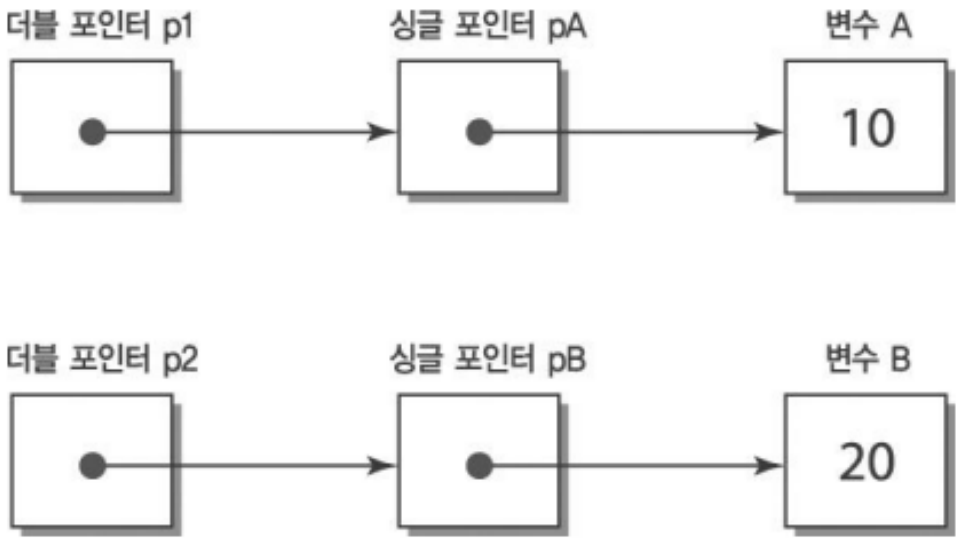
```
void pswap(int *p1, int *p2)
{
    int *temp;
    temp=p1;
    p1=p2;
    p2=temp;
}
```



구현 사례 2 : 효과 없는 swap 함수의 호출

```
/* ptr_swap2c */
#include <stdio.h>
void pswap(int **p1, int **p2);
int main(void)
{
    int A=10, B=20;
    int *pA, *pB;
    pA=&A, pB=&B;

    pswap(&pA, &pB);

    printf("pA가 가리키는 변수 : %d \n", *pA);
    printf("pB가 가리키는 변수 : %d \n", *pB);
    return 0;
}
```

```
void pswap(int **p1, int **p2)
{
    int *temp;
    temp=*p1;
    *p1=*p2;
    *p2=temp;
}
```



질의 및 응답

참고문헌

- 천정아,『Core C Programming』, 연두에디션(2019)
- C가 보이는 그림책, ANK Co, Ltd , 성안당 (2018)
- Greg Perry, Dean Miller『어서와 C언어는 처음이지』, 천인국 옮김, 인피니티북스(2015)
- KELLEY (역 : 김명호 외),『A Book on C』, 홍릉과학출판사 (2003)
- 윤성우,『 열혈 C 프로그래밍』, 오렌지미디어
- 천인국,『 쉽게 풀어쓴 C언어 Express』, 생능출판사
- 서현우,『 뇌를 자극하는 C 프로그래밍』, 한빛미디어
- 강성수,『 쾌도난마 C프로그래밍』, 북스홀릭
- 고응남,『 C프로그래밍 기초와 응용실습』, 정익사