

컴퓨터 프로그래밍2

포인터 응용

장서윤 pineai@cnu.ac.kr

2중 포인터와 배열 포인터

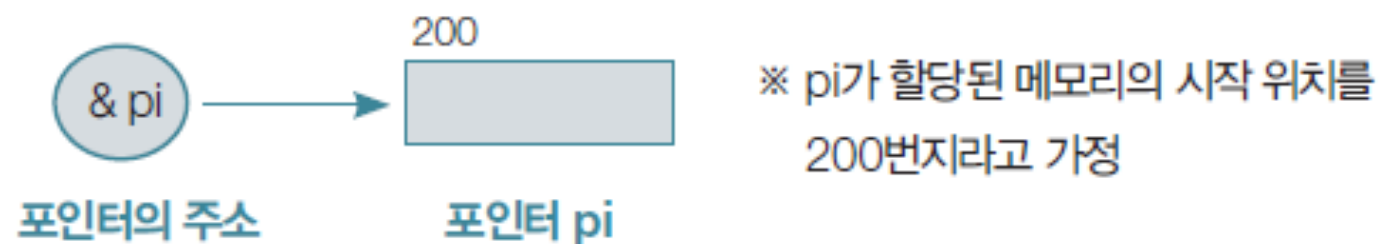
❖ 이중 포인터와 배열의 용도

표 15-1 2중 포인터와 배열 포인터의 용도

구분	기능	설명
2중 포인터	선언 방법	<code>int **p;</code>
	사용 예 1	포인터를 교환하는 함수의 매개변수로 사용
	사용 예 2	포인터 배열을 처리하는 함수의 매개변수로 사용
배열 포인터	선언 방법	<code>int (*pa)[4];</code>
	의미	int형 변수 4개짜리 1차원 배열을 가리킨다.
	사용 예	2차원 배열의 배열명을 받는 함수의 매개변수에 사용

2중 포인터 개념

- ▶ 포인터도 메모리에 저장 공간 갖는 하나의 변수
 - ▶ 주소 연산으로 주소 구할 수 있음
 - ▶ 이 주소를 저장하는 포인터가 2중 포인터
- ▶ 2중 포인터에 간접참조 연산 수행
 - ▶ 가리키는 대상인 포인터 쓸 수 있음



2중 포인터 개념

예제 15-1 포인터와 2중 포인터의 관계

```

1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int a = 10;        // int형 변수의 선언과 초기화
6.     int *pi;           // 포인터 선언
7.     int **ppi;         // 2중 포인터 선언
8.
9.     pi = &a;           // int형 변수의 주소를 포인터에 저장
10.    ppi = &pi;         // 포인터의 주소를 2중 포인터에 저장
11.
12.    printf("-----\n");
13.    printf("변수  변수값  &연산  *연산  **연산\n");
14.    printf("-----\n");
15.    printf(" a%10d%10u\n", a, &a);
16.    printf(" pi%10u%10u%10d\n", pi, &pi, *pi);
17.    printf(" ppi%10u%10u%10u%10u\n", ppi, &ppi, *ppi, **ppi);
18.    printf("-----\n");
19.
20.    return 0;
21. }
```

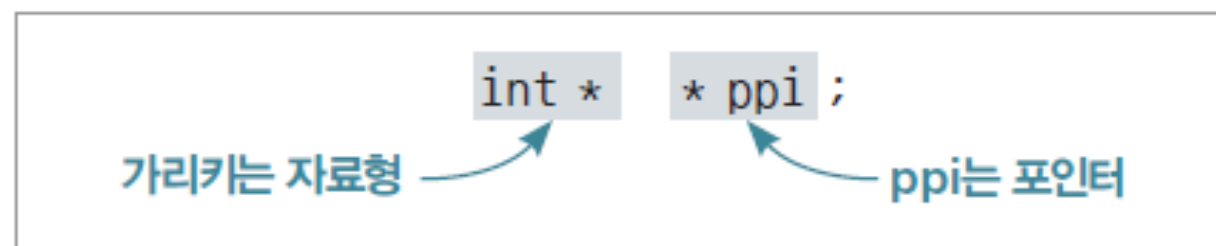
실행 결과	변수	변수값	&연산	*연산	**연산
	a	10	2160944		
	pi	2160944	2160932	10	
	ppi	2160932	2160920	2160944	10

```

* *pp = 10
* p = 10
a = 10
```

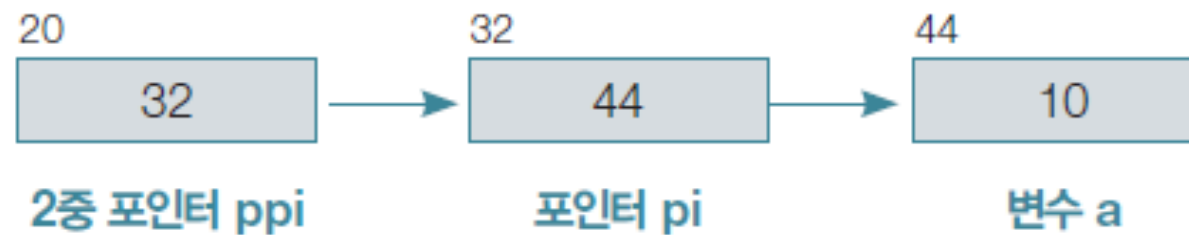
2중 포인터 개념

- ▶ 2중 포인터는 별 (*) 2개 붙여 선언
 - ▶ 첫 번째 별은 가리키는 변수의 자료형
 - ▶ 두 번째 별은 자신의 자료형
- ▶ 첫 번째 별은 ppi가 가리키는 자료형이 포인터
- ▶ 두 번째 별은 ppi 자신이 포인터
- ▶ 2중 포인터 선언하여 메모리에 저장 공간 할당
 - ▶ 그 이후에는 변수명으로 사용



2중 포인터 개념

- ▶ 변수 a의 주소 pi에 저장
 - ▶ 주소는 설명의 편의 위해 실행 결과 마지막 2자리 표시
 - ▶ 값은 프로그램이 실행될 때, 메모리의 상황에 따라 다를 수 있음

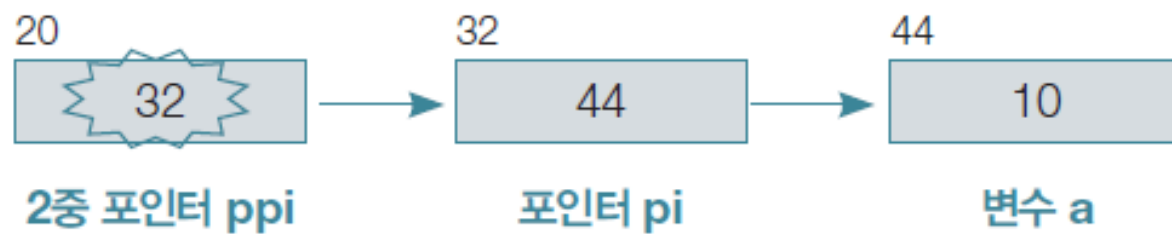


9. `pi = &a;` // int형 변수의 주소를 포인터에 저장

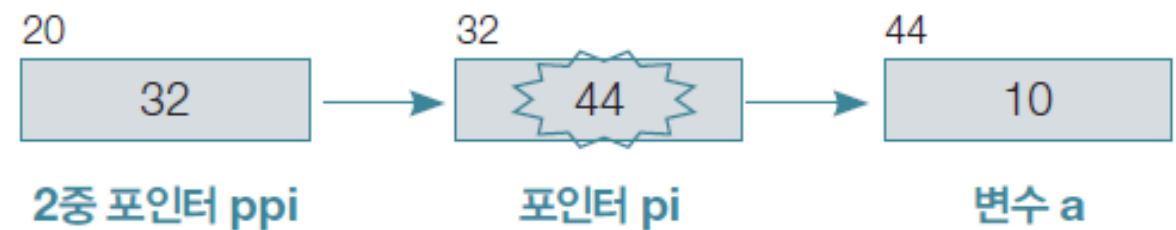
2중 포인터 개념

▶ 변수명 쓰면 값, &연산 하면 주소, *연산은 화살표 따라감

- [변수명 사용] 17행의 ppi - 2중 포인터 ppi값



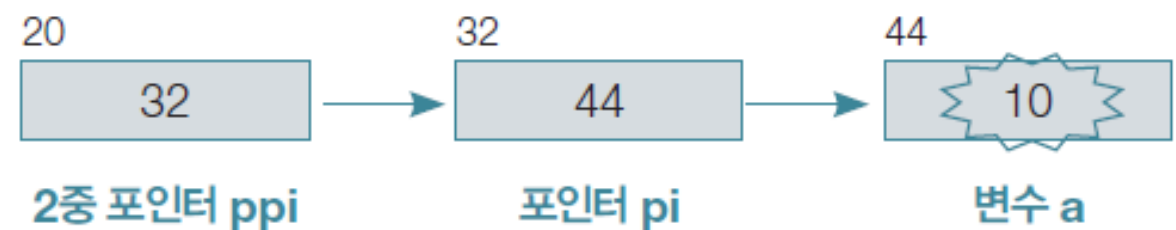
- [* 연산자 사용] 17행의 *ppi - 2중 포인터 ppi가 가리키는 변수의 값



- [& 연산자 사용] 16행의 &pi - 포인터 pi의 주소



- 17행의 **ppi - 2중 포인터 ppi가 가리키는 포인터가 가리키는 변수의 값

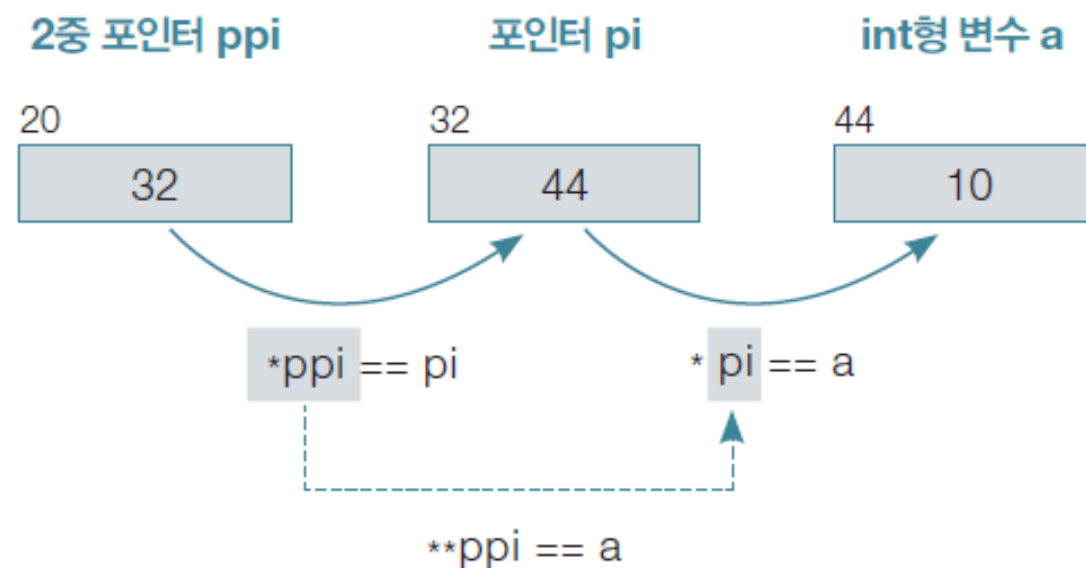


```
16.    printf(" pi%10u%10u%10d\n", pi, &pi, *pi);
```

```
17.    printf("ppi%10u%10u%10u%10u\n", ppi, &ppi, *ppi, **ppi);
```

2중 포인터 개념

- ▶ 2중 포인터 ppi로 변수 a값을 사용하기 위해서
 - ▶ 간접참조 연산자 두 번 써야
 - ▶ ppi가 가리키는 것은 포인터 pi
 - ▶ ppi에 간접참조 연산 수행
 - ▶ pi에 저장된 주소값 구함
 - ▶ pi가 가리키는 변수 a값을 쓰려면 간접참조 연산자 한 번 더 사용



2중 포인터 개념

- ▶ 2중 포인터도 가리키는 포인터의 형태에 맞춰 선언
 - ▶ 저장할 주소가 어떤 자료형의 주소인지 파악
- ▶ Ex) 변수와 포인터 선언

```
double a = 3.5;  
double *pi = &a;
```


- ▶ pi가 (double *)형 변수
- ▶ &pi는 (double *)형의 주소

2중 포인터 개념

- ▶ (double *)형 가리키는 이중 포인터 선언

```
double **ppi;
```

```
ppi = &pi ;    // 포인터의 주소를 2중 포인터에 대입
```



가리키는 자료형이 (double *)형으로 같다!

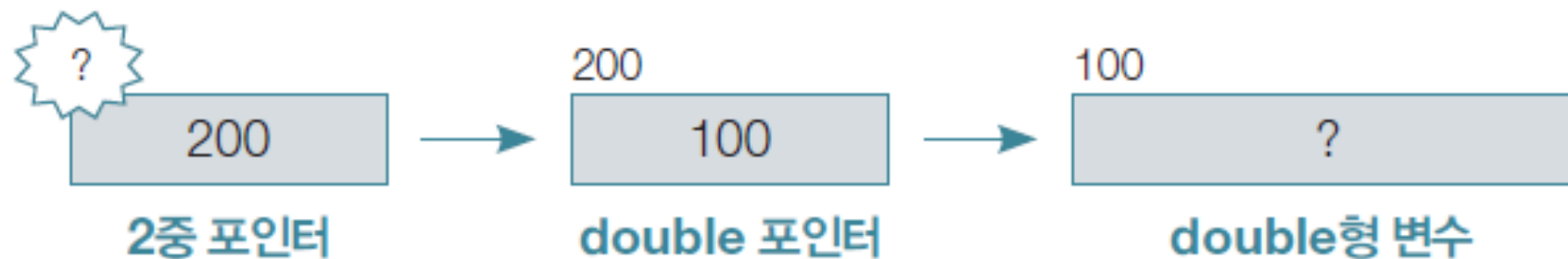
2중 포인터 개념

- ▶ 주소 상수에는 주소 연산자 쓸 수 없음
 - ▶ 포인터는 변수이므로 주소 연산자 사용하여 그 주소를 구할 수 있지만 상수인 주소에는 주소 연산자를 쓸 수 없음

```
int a;  
int *pi = &a;      // 주소를 포인터에 저장  
  
&pi;               // 포인터에 주소 연산자 사용 가능  
&(&a);             // a의 주소에 다시 주소 연산자 사용 불가능
```

2중 포인터 개념

- ▶ 2중 포인터의 주소는 3중 포인터에 저장
 - ▶ 2중 포인터도 변수이므로 주소 연산자 사용하면 주소를 구할 수 있음
 - ▶ 2중 포인터가 가리키는 자료형이 double 포인터일 때 2중 포인터를 가리키는 3중 포인터

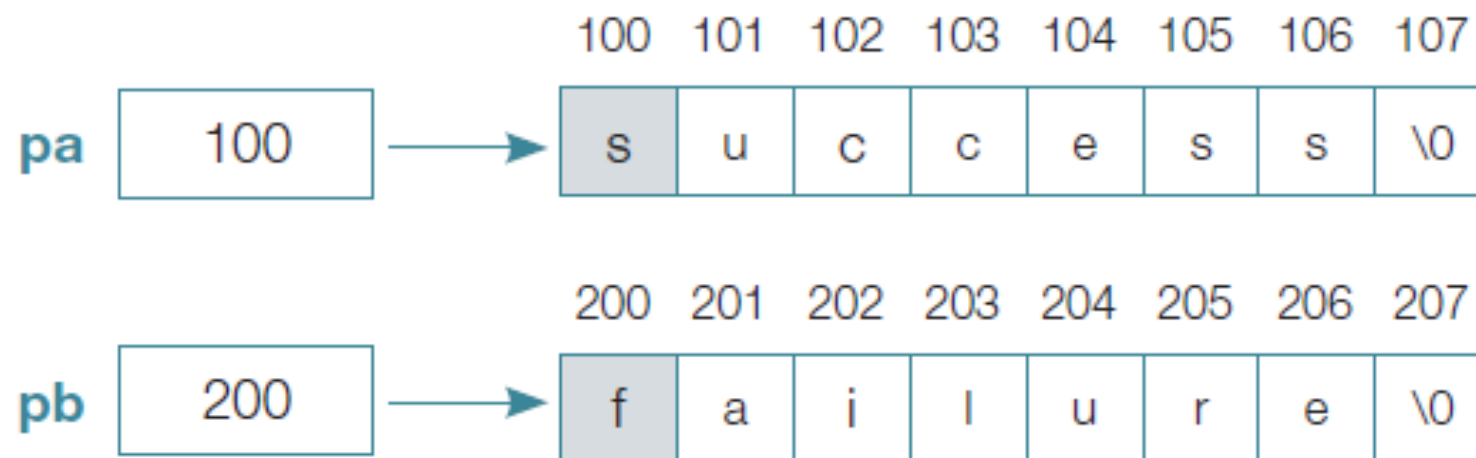


- ▶ 4중 이상포인터는 가독성 떨어지므로 사용하지 않음
- ▶ 다중포인터 -> 2중 이상의 포인터

2중 포인터 활용 예

- ▶ 포인터의 값 바꾸는 함수의 매개변수에 사용
- ▶ Ex) 다음과 같이 2개의 포인터가 문자열 연결

```
char *pa = "success";  
char *pb = "failure";
```



```

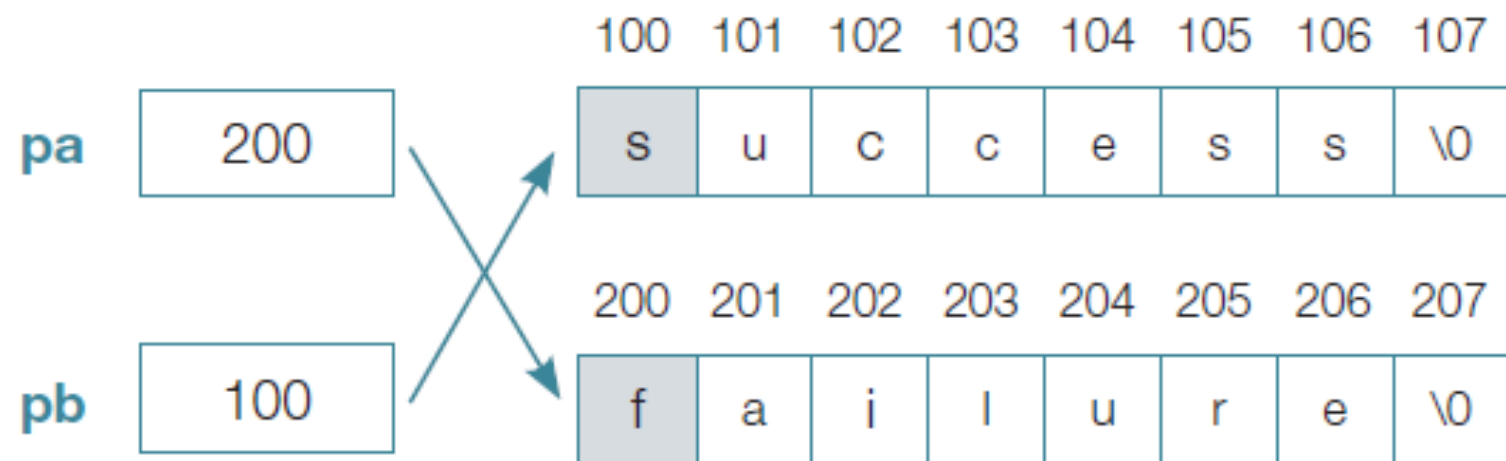
1. #include <stdio.h>
2.
3. void swap_ptr(char **ppa, char **ppb); .....
4.
5. int main(void)
6. {
7.     char *pa = "success";
8.     char *pb = "failure";
9.
10.    printf("pa -> %s, pb -> %s\n", pa, pb);    // 바꾸기 전에 문자열 출력
11.    swap_ptr(&pa, &pb);                        // 함수 호출
12.    printf("pa -> %s, pb -> %s\n", pa, pb);    // 바꾼 후에 문자열 출력
13.
14.    return 0;
15. }
16.
17. void swap_ptr(char **ppa, char **ppb)
18. {
19.     char *pt;
20.
21.     pt = *ppa;
22.     *ppa = *ppb;
23.     *ppb = pt;
24. }

```

실행 결과
 pa -> success, pb -> failure
 pa -> failure, pb -> success

2중 포인터 활용 예

- ▶ 문자열 바꿔 출력하지만 문자열 자체는 바꾸지 않음
 - ▶ 문자열 연결하는 포인터의 값 바꾸면 연결 상태 바뀜
- ▶ 이후에 포인터 사용하면 마치 문자열 바꾼 것처럼 사용



2중 포인터 활용 예

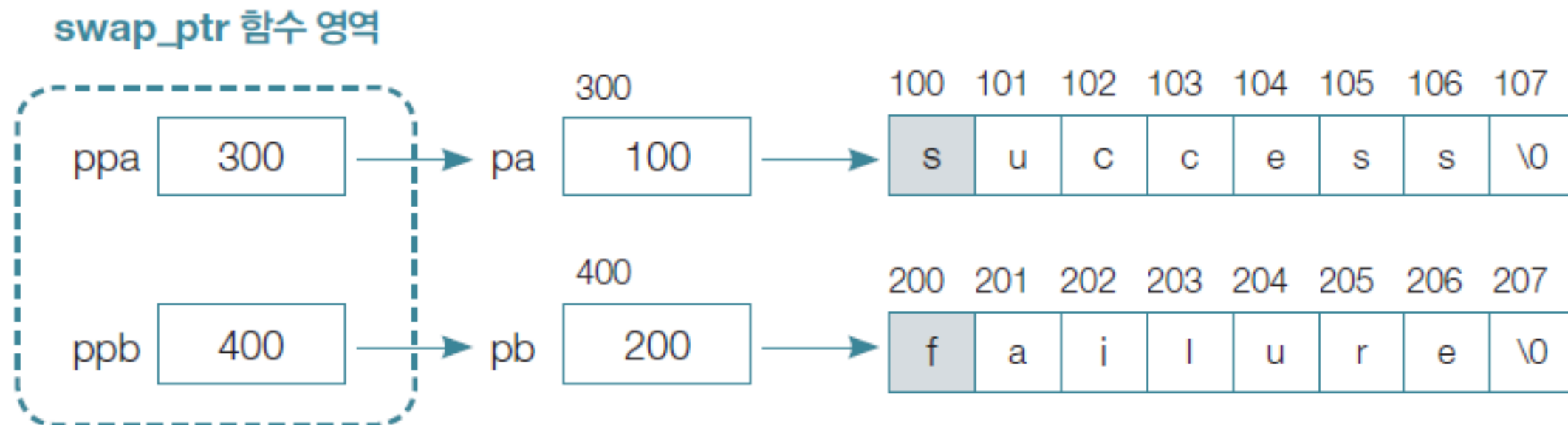
▶ 함수 선언과 메모리 영역 표시 결과

- 함수 호출

```
swap_ptr(&pa, &pb); // pa, pb의 주소를 인수로 주고 호출
```

- 함수 선언 pa, pb의 형태가 char *이므로 char *형의 주소가 넘어오게 되므로 그 주소를 저장할 *형으로 받아줘야 한다.

```
void swap_ptr(char **ppa, char **ppb); // 매개변수로 2중 포인터 사용
```

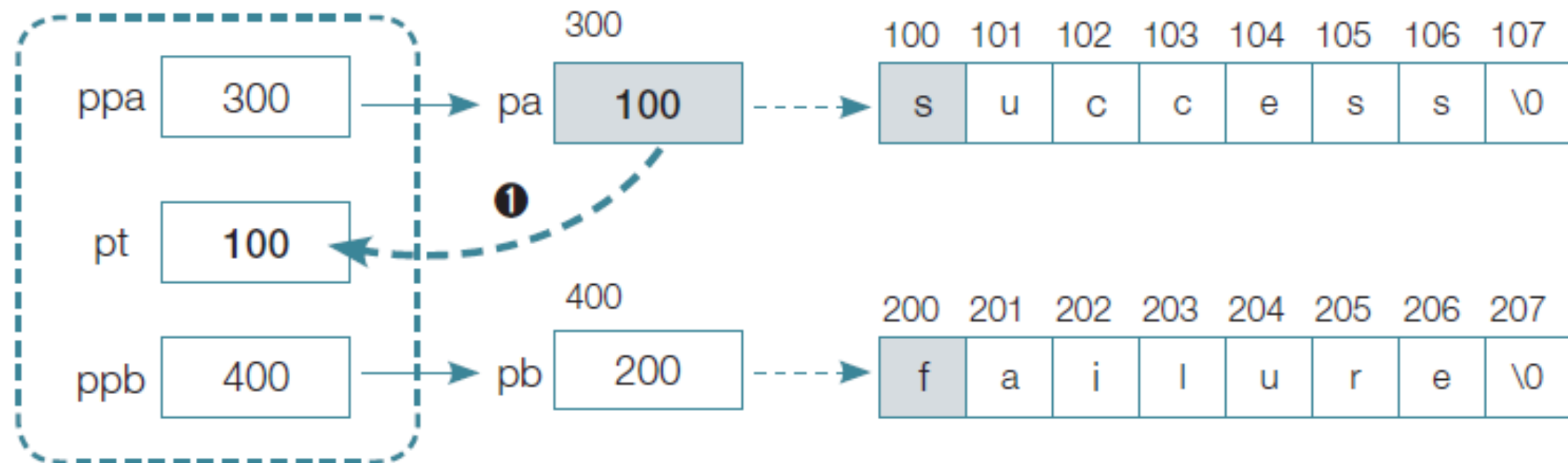


2중 포인터 활용 예

*ppa = pa

1. pt = *ppa; // ppa가 가리키는 pa의 값을 pt에 저장

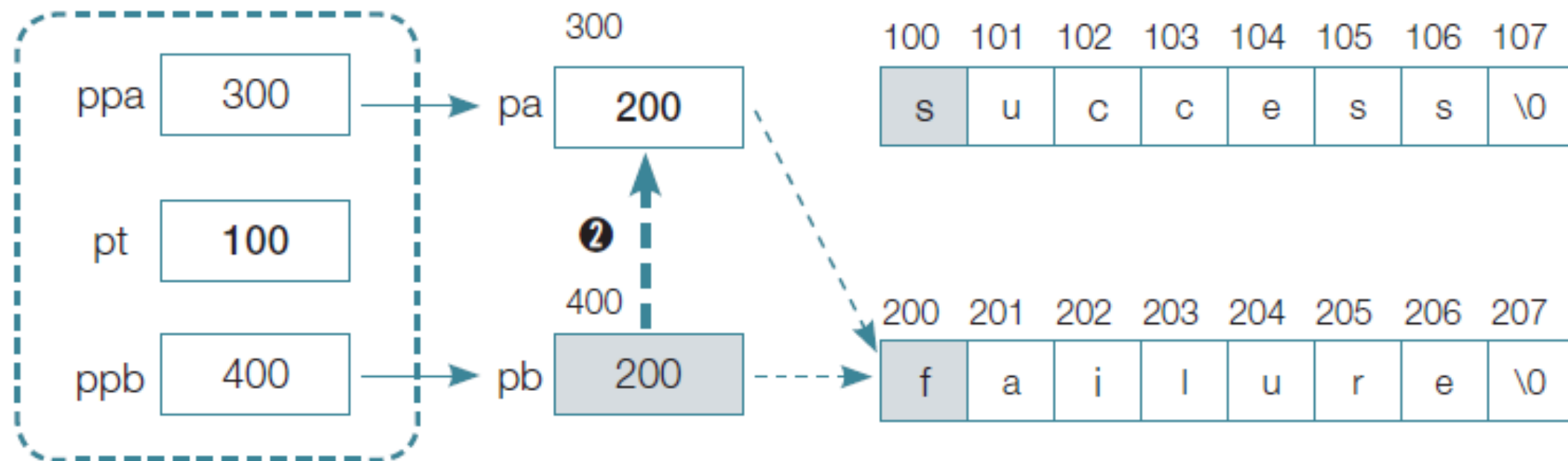
swap_ptr 함수 영역



2중 포인터 활용 예

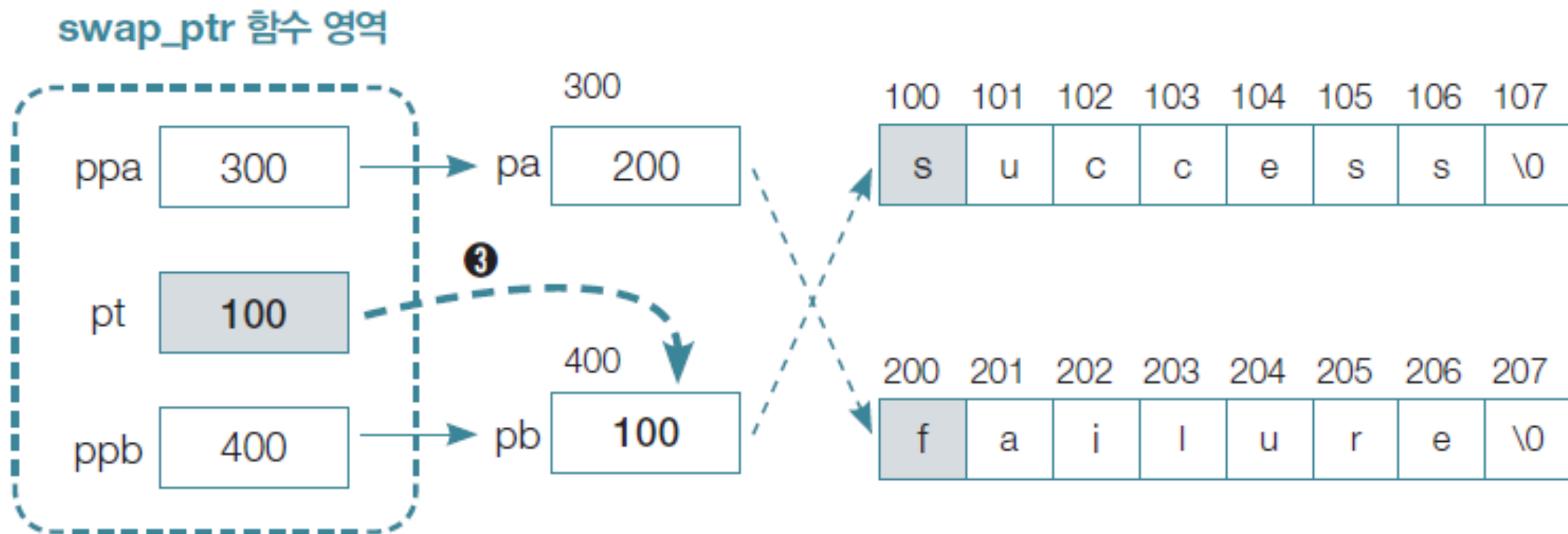
2. `*ppa = *ppb ;` // ppb가 가리키는 pb의 값을 ppa가 가리키는 pa에 저장

swap_ptr 함수 영역



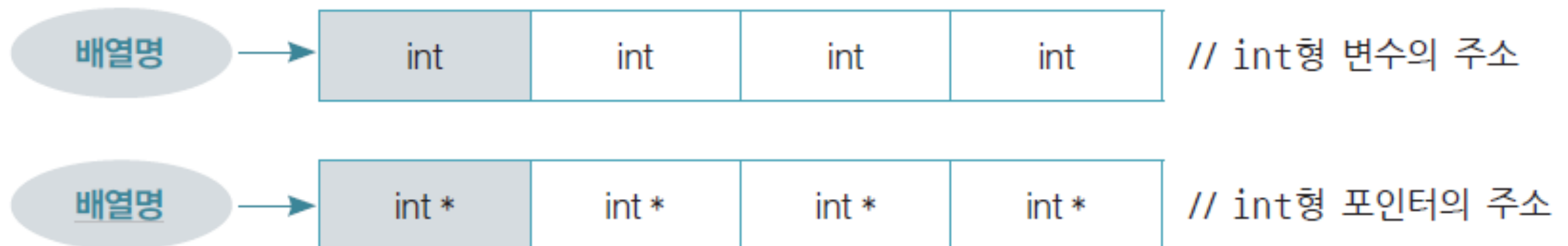
2중 포인터 활용 예

3. `*ppb = pt;` // pt의 값을 ppb가 가리키는 pb에 저장



2중 포인터 활용 예

- ▶ 2중 포인터는 포인터 배열 매개변수로 받는 함수에도 사용
 - ▶ 배열명은 첫 번째 배열 요소의 주소
 - ▶ int형 배열의 이름은 int형 변수의 주소
 - ▶ int형 포인터 배열의 이름은 int형 포인터의 주소
 - ▶ 포인터 배열의 배열명은 2중 포인터에 저장



예제 15-3 포인터 배열의 값을 출력하는 함수

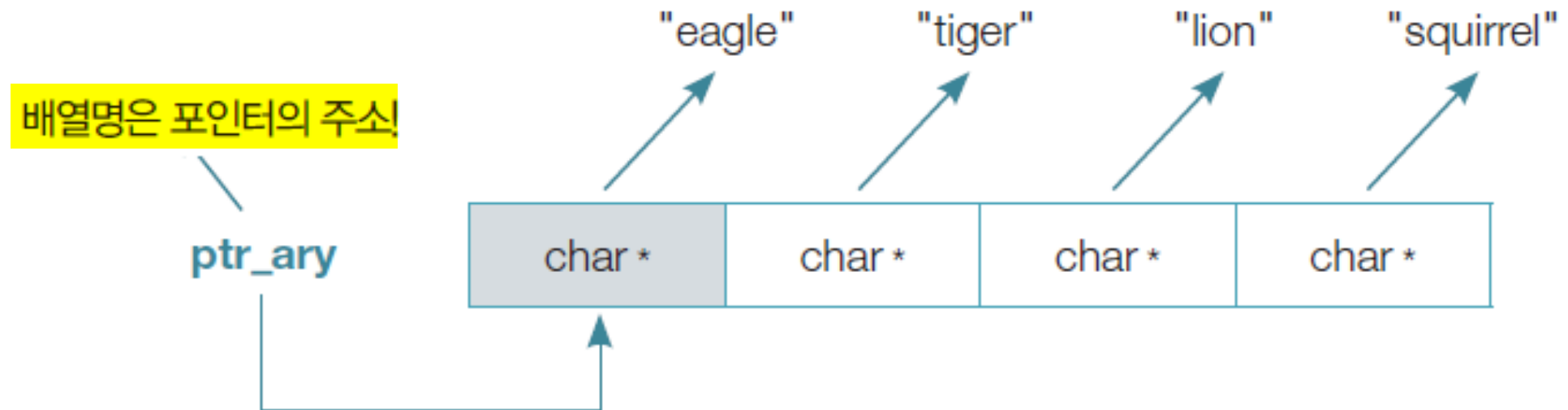
```
1. #include <stdio.h>
2.
3. void print_str(char **pps, int cnt);
4.
5. int main(void)
6. {
7.     char *ptr_ary[] = {"eagle", "tiger", "lion", "squirrel"}; // 초기화
8.     int count; // 배열 요소 수를 저장할 변수
9.
10.    count = sizeof(ptr_ary) / sizeof(ptr_ary[0]); // 배열 요소의 수 계산
11.    print_str(ptr_ary, count); // 배열명과 배열 요소 수를 주고 호출
12.
13.    return 0;
14. }
15.
16. void print_str(char **pps, int cnt) // 매개변수로 2중 포인터 사용
17. {
18.     int i;
19.
20.     for(i = 0; i < cnt; i++) // 배열 요소 수만큼 반복
21.     {
22.         printf("%s\n", pps[i]); // 2중 포인터를 배열명처럼 사용
23.     }
24. }
```

실행
결과

```
eagle
tiger
lion
squirrel
```

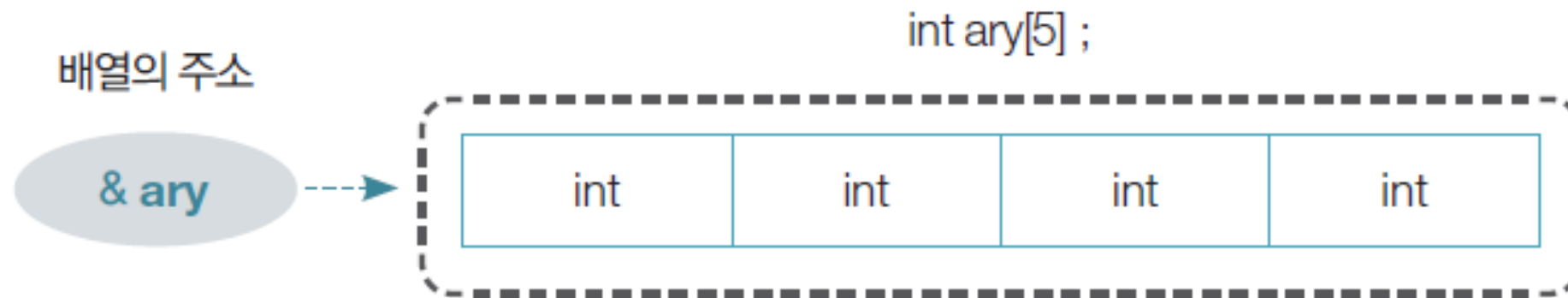
2중 포인터 활용 예

- ▶ 포인터가 배열명을 저장하면 배열명처럼 쓸 수 있으므로 함수 안에서는 매개변수를 배열명처럼 사용하여 문자열 출력



배열 요소의 주소와 배열의 주소 (배열 포인터)

▶ 배열에 주소 연산자 사용하면 배열을 가리키는 주소



배열명이 갖는 2가지 의미

- 배열의 첫 번째 요소의 시작주소 - 주소상수
- 배열전체의 저장공간을 나타내는 논리적인 변수

배열 요소의 주소와 배열의 주소

예제 15-4 주소로 쓰이는 배열과 배열의 주소 비교

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int ary[5];
6.
7.     printf("ary의 값 : %u\n", ary);           // 주소로서의 배열명의 값
8.     printf("ary의 주소 : %u\n", &ary);        // 배열의 주소
9.
10.    printf("ary + 1 : %u\n", ary + 1);
11.    printf("&ary + 1 : %u\n", &ary + 1);
12.
13.    return 0;
14. }
```

실행
결과

```
ary의 값 : 3275140
ary의 주소 : 3275140
ary + 1 : 3275144
&ary + 1 : 3275160
```


배열 요소의 주소와 배열의 주소

▶ 배열의 주소에 정수 더하면 배열 전체의 크기 곱해서 더함

- 배열의 정수 연산

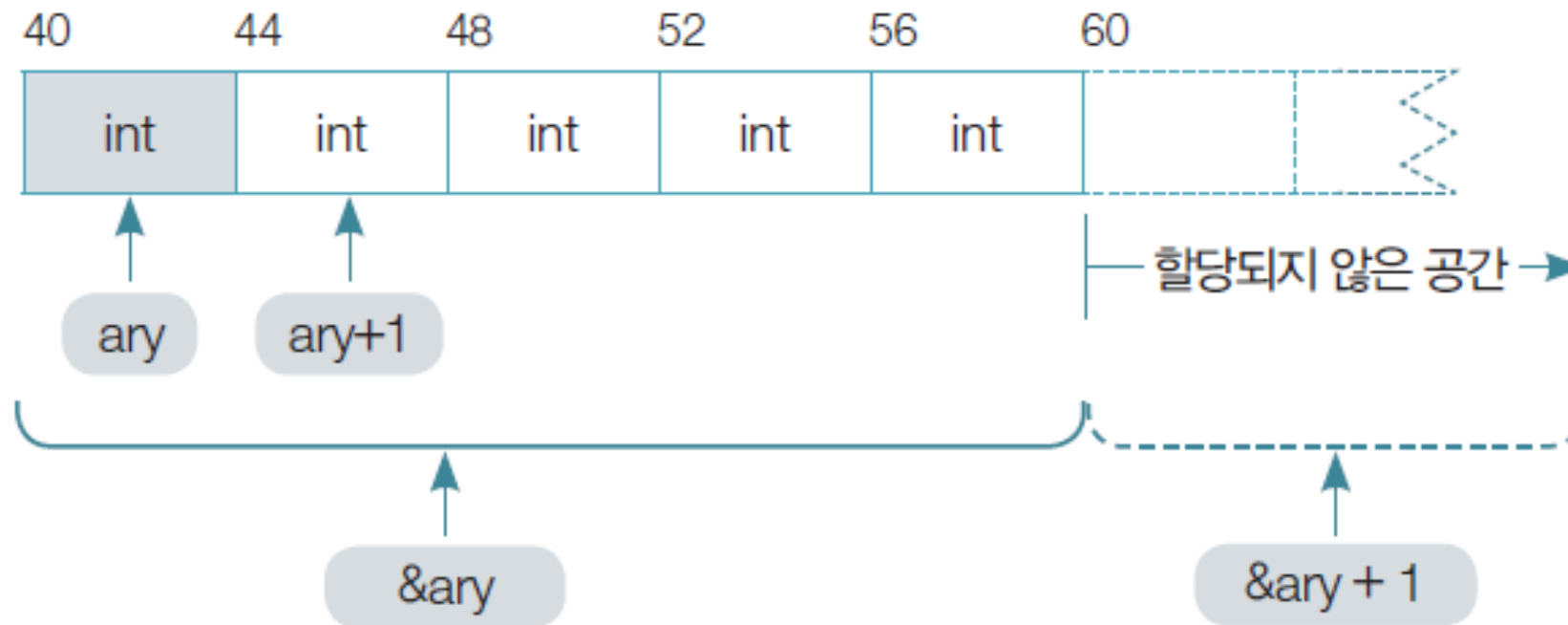
$\text{ary} + 1 \rightarrow 3275140 + (1 * \text{sizeof}(\text{ary}[0])) \rightarrow 3275140 + (1 * 4) \rightarrow 3275144$

- 배열의 주소에 정수 연산

$\&\text{ary} + 1 \rightarrow 3275140 + (1 * \text{sizeof}(\text{ary})) \rightarrow 3275140 + (1 * 20) \rightarrow 3275160$

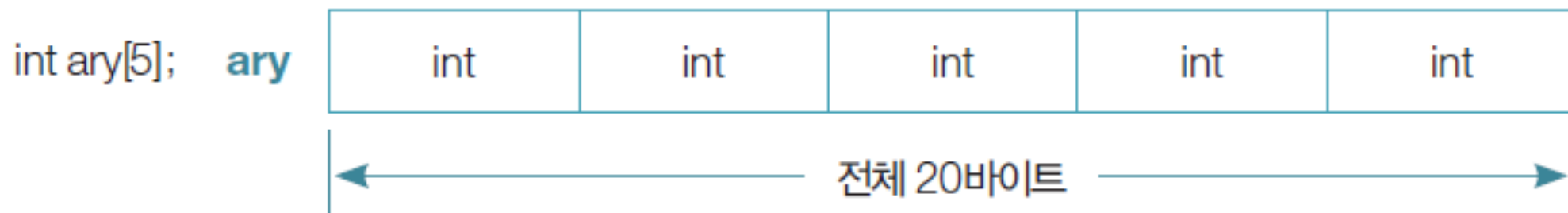
배열 요소의 주소와 배열의 주소

- ▶ 1차원 배열에서 배열의 주소를 구하고 정수 더하는 연산 가능
 - ▶ 배열의 주소에 정수를 더하면 배열 할당된 메모리 영역 벗어나므로 특별한 경우가 아니면 사용하지 않음
- ▶ 배열의 주소는 주로 2차원 이상의 배열에서 사용



배열 요소의 주소와 배열의 주소

- ▶ 배열도 일반 변수처럼 크기와 형태에 대한 정보 가짐
- ▶ Ex) 배열 `ary`는 크기가 20바이트며 'int형 변수 5개의 배열'이란 자료형의 정보를 가짐



- ▶ 배열 전체가 하나의 변수와 같은 역할
 - ▶ 변수에 사용하는 연산자 배열에도 쓸 수 있음
 - ▶ `sizeof (ary)` - 배열 전체의 크기 계산
 - ▶ `&ary` - 배열 전체 시작 주소

배열 요소의 주소와 배열의 주소

- ▶ 배열은 논리적인 변수
 - ▶ 일반 변수처럼 대입 연산자 왼쪽에 사용하는 것 불가능

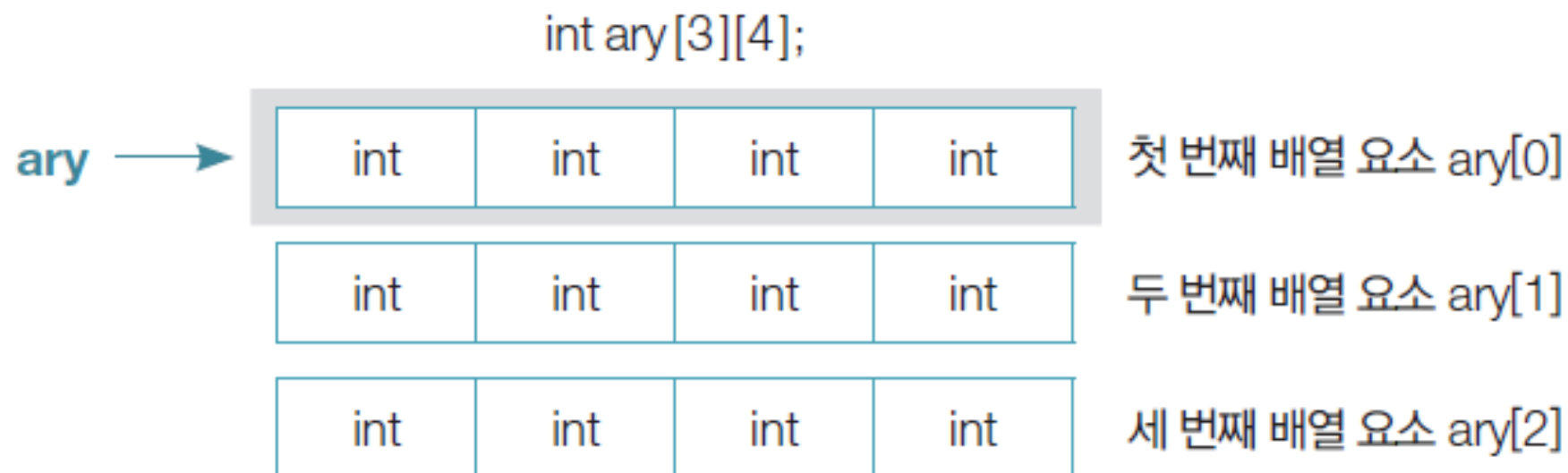
```
int ary[5];
```

```
ary = 10; // ary 배열의 5개 요소 중에 어떤 요소에 저장할지 알 수 없음
```

 ※ 주의 : 이렇게 대입할 수 없습니다!

2차원 배열과 배열 포인터

- ▶ 2차원 배열은 1차원 배열로 만든 배열
 - ▶ 논리적인 배열 요소가 1차원 배열
- ▶ 배열명은 첫 번째 요소의 주소이므로 다음과 같은 정리 가능
 - ▶ 2차원 배열의 이름은 1차원 배열의 주소
 - ▶ 배열 가리키는 포인터에 저장



예제 15-5 배열 포인터로 2차원 배열의 값 출력

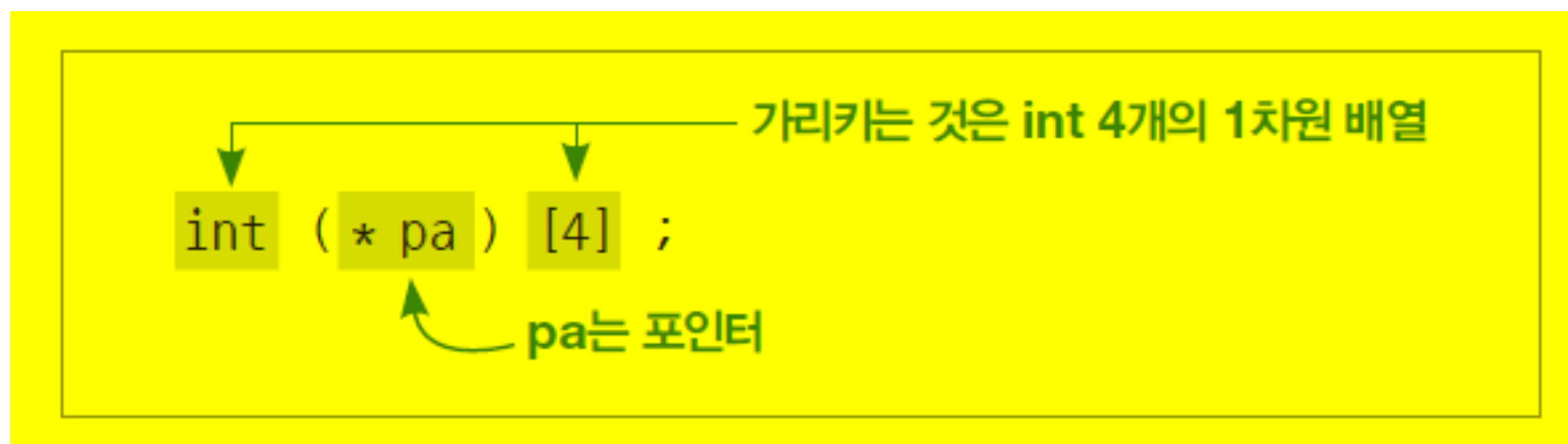
```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int ary[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
6.     int (*pa)[4];           // int형 변수 4개의 배열을 가리키는 배열 포인터
7.     int i, j;
8.
9.     pa = ary;
10.    for(i = 0; i < 3; i++)
11.    {
12.        for(j = 0; j < 4; j++)
13.        {
14.            printf("%5d", pa[i][j]); // pa를 2차원 배열처럼 사용
15.        }
16.        printf("\n");
17.    }
18.
19.    return 0;
20. }
```

실행
결과

1	2	3	4
5	6	7	8
9	10	11	12

2차원 배열과 배열 포인터

- ▶ 2차원 배열의 이름을 저장할 배열 포인터
 - ▶ 변수명 앞에 별 (*) - 포인터임을 표시하고 괄호로 묶음
 - ▶ 양 옆에 가리키는 배열의 형태를 나누어 적음
- ▶ 선언된 배열 포인터는 메모리에 저장 공간 확보
 - ▶ 그 이후부터는 이름으로 사용
- ▶ Ex) ary가 가리키는 첫 번째 부분배열은 (int [4]) 형
 - ▶ 다음과 같이 pa 선언합니다. 괄호가 없으면 포인터 배열이 되므로 주의



2차원 배열과 배열 포인터

예제 15-6 2차원 배열의 값을 출력하는 함수

```
1. #include <stdio.h>
2.
3. void print_ary(int (*)[4]);
4.
5. int main(void)
6. {
7.     int ary[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
8.
9.     print_ary(ary);           // 배열명을 인수로 주고 함수 호출
10.
11.     return 0;
12. }
13.
```


2차원 배열과 배열 포인터

```
14. void print_ary(int (*pa)[4])           // 매개변수는 배열 포인터
15. {
16.     int i, j;
17.
18.     for(i = 0; i < 3; i++)
19.     {
20.         for(j = 0; j < 4; j++)
21.         {
22.             printf("%5d", pa[i][j]);    // pa를 2차원 배열처럼 사용
23.         }
24.         printf("\n");
25.     }
26. }
```

실행 결과	1	2	3	4
	5	6	7	8
	9	10	11	12

2차원 배열과 배열 포인터

- ▶ 2차원 배열에는 두 가지 의미의 배열 요소
 - ▶ 2차원 배열의 요소는 1차원 배열이지만 실제로 데이터가 저장되는 공간은 1차원 배열의 요소
- ▶ 2차원 배열에서 ‘배열 요소’는 논리적으로는 1차원의 부분배열, 물리적으로는 실제 데이터 저장하는 부분배열의 요소
 - ▶ Ex) `int ary [3] [4];`
 - ▶ 2차원 배열 `ary`의 논리적 배열 요소 개수는? 3개
 - ▶ 2차원 배열 `ary`의 물리적 배열 요소 개수는? 12개

2차원 배열의 요소를 참조하는 원리

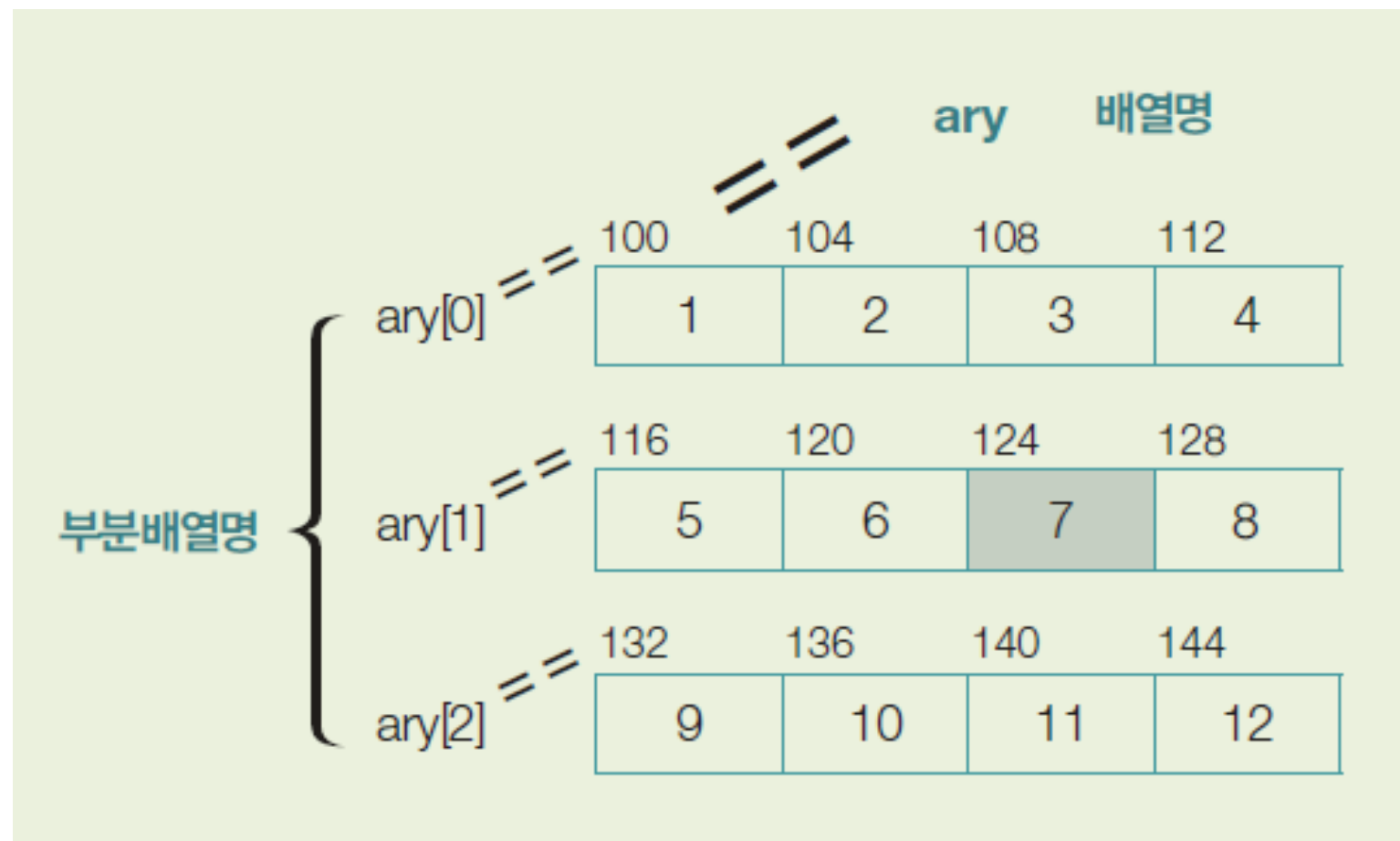
▶ 2차원 배열

- ▶ 모든 저장 공간이 메모리에 연속으로 할당 (1차원 배열과 동일)
- ▶ 2차원의 논리적 공간으로 사용
 - ▶ 배열명이 1차원 배열의 주소로 1차원 배열 전체 가리킴
 - ▶ 배열 포인터 쓰면 1차원의 물리적 공간을 2차원의 논리적 구조로 사용할 수 있음

2차원 배열의 요소를 참조하는 원리

▶ Ex) 2차원 배열 `int ary[3][4];`

▶ 물리적 요소 참조 과정



2차원 배열의 요소를 참조하는 원리

- ▶ Ex) 2차원 배열 `int ary[3][4];`
 - ▶ 7번째 물리적 요소 참조 과정
 - ▶ 7번째 물리적 요소는 두 번째 부분배열 `ary[1]`에 속하므로 먼저 `ary[1]`의 시작 위치 구해야 함
 - ▶ 2차원 배열 `ary`는 첫 번째 부분배열의 주소
 - ▶ `ary`에 1을 더하면 두 번째 부분배열 `ary[1]`의 주소를 구할 수 있음

`ary + 1` → `100 + (1 * sizeof(ary[0]))` → `100 + (1 * 16)` → `116`

↖ `ary`가 가리키는 첫 번째 부분배열의 크기

2차원 배열의 요소를 참조하는 원리

- ▶ Ex) 2차원 배열 `int ary[3][4];`
 - ▶ 116번지에 2를 더해 7번째 물리적 배열 요소의 위치?
 - ▶ $(ary + 1) + 2$ 의 값은 $ary + 3 \rightarrow$ 148번지
 - ▶ $ary + 1$ 의 결과인 116번지 - 두 번째 부분배열 전체 가리키는 주소
 - ▶ 116번지에 간접 참조 연산자 사용
 - ▶ 두 번째 부분배열 구하는 과정 필요

```
*(ary + 1) → ary[1]    // 두 번째 부분배열
```

2차원 배열의 요소를 참조하는 원리

- ▶ Ex) 2차원 배열 `int ary[3][4];`
 - ▶ 부분배열명 `ary[1]` - 두 번째 부분배열의 첫 번째 요소 주소
 - ▶ `*(ary+1)`의 연산 결과는 다섯 번째 물리적 요소 주소
 - ▶ 여기에 2 더하면 124번지

`*(ary + 1) + 2` → `*(ary + 1) + (2 * sizeof(ary[1][0]))` → `116 + (2 * 4)` → `124`

↖ 두 번째 부분배열의 첫 번째 배열 요소의 크기

2차원 배열의 요소를 참조하는 원리

- ▶ Ex) 2차원 배열 `int ary[3][4];`
 - ▶ 연산의 결과값 124번지 - 7번째 물리적 배열 요소의 주소
 - ▶ 그 요소 쓰기 위해 간접참조 연산 수행

`*(*ary + 1) + 2) → ary[1][2]` // 두 번째 부분배열의 세 번째 배열 요소

2차원 배열의 요소를 참조하는 원리

- ▶ Ex) 2차원 배열 `int ary[3][4];` 에서 같은 값 가지는 주소들
 - ▶ 1. `&ary` // 2차원 배열 전체의 주소
 - ▶ 2. `ary` // 첫 번째 부분배열의 주소
 - ▶ 3. `&ary [0]` // 첫 번째 부분배열의 주소
 - ▶ 4. `ary [0]` // 첫 번째 부분배열의 첫 번째 배열 요소의 주소
 - ▶ 부분배열로 논리적으로 변수의 기능을 가짐
 - ▶ 5. `&ary [0][0]` // 첫 번째 부분배열의 첫 번째 배열 요소의 주소
 - ▶ 단지 주소
- ▶ 주소는 가리키는 자료형이 다르면 같다고 할 수 없음
 - ▶ 2, 3번과 4, 5번만 서로 같음
- ▶ 2번은 배열이고 3번은 단순한 주소
 - ▶ 배열은 주소뿐만 아니라 논리적으로 변수의 기능
 - ▶ `sizeof` 연산 수행하면 크기가 서로 다름

함수 포인터와 VOID 포인터

- ❖ 함수 이름은 함수가 있는 메모리의 주소
- ❖ 함수 포인터와 void 포인터의 용도

표 15-2 함수 포인터와 void 포인터의 용도

구분	기능	설명
함수 포인터	선언 방법	<code>int (*fp)(int, int);</code>
	함수의 호출	<code>fp(10, 20);</code>
	용도	함수명을 대입하여 호출 함수를 결정한다.
void 포인터	선언 방법	<code>void *vp;</code>
	의미	가리키는 자료형에 대한 정보가 없다.
	용도	임의의 주소를 받는 함수의 매개변수에 사용한다.

함수 포인터의 개념

- ▶ 함수 포인터 사용하기 위해 가장 중요한 것
 - ▶ 함수명의 의미를 파악하는 것
- ▶ 함수명 - 함수의 정의 있는 메모리의 시작 위치
 - ▶ 함수명이 주소이므로 포인터에 저장
 - ▶ 함수 다양한 방식으로 호출 가능

예제 15-7 함수 포인터를 사용한 함수 호출

```
1. #include <stdio.h>
2.
3. int sum(int, int);           // 함수 선언
4.
5. int main(void)
6. {
7.     int (*fp)(int, int);     // 함수 포인터 선언
8.     int res;                 // 반환값을 저장할 변수
9.
10.    fp = sum;                 // 함수명을 함수 포인터에 저장
11.    res = fp(10, 20);         // 함수 포인터로 함수 호출
12.    printf("result : %d\n", res); // 반환값 출력
13.
14.    return 0;
15. }
16.
17. int sum(int a, int b)
18. {
19.     return (a + b);
20. }
```

.....

실행
결과 result : 30

함수 포인터의 개념

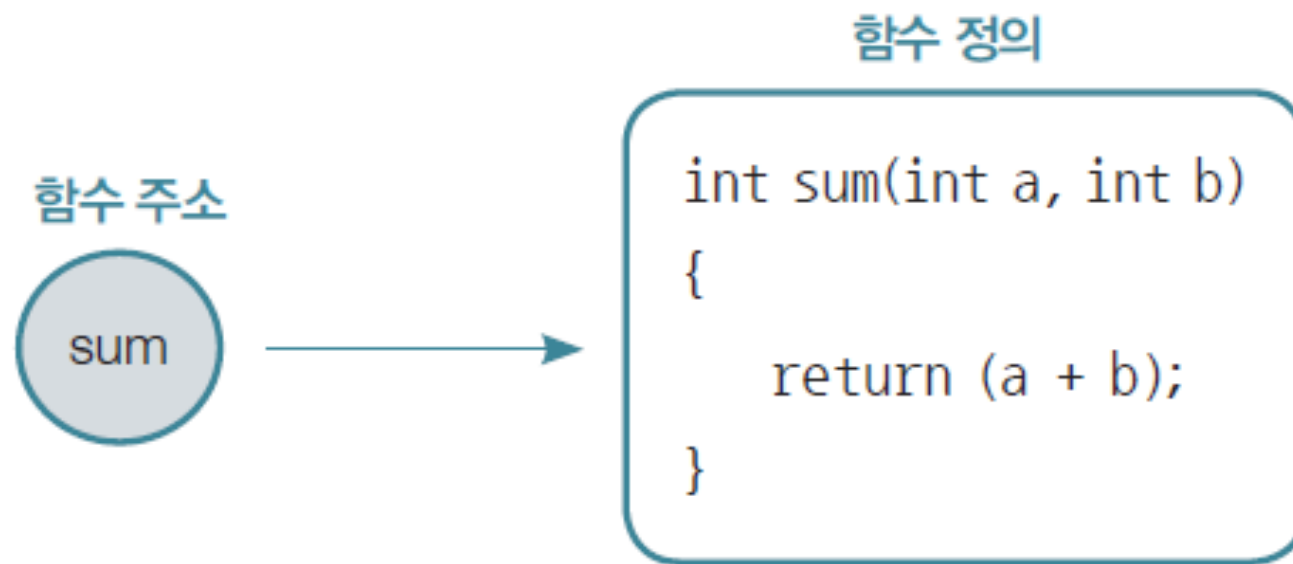
➤ sum 함수 정의

➤ 프로그램이 컴파일 되면 함수도 실행 파일의 한 부분 차지

➤ 프로그램이 실행되면 함수도 메모리에 올려짐

➤ 메모리에 올려진 함수 실행 위해서는 그 위치를 알아야

➤ 함수명이 그 주소로 바뀜 - 함수 호출 할 때 함수명 사용



함수 포인터의 개념

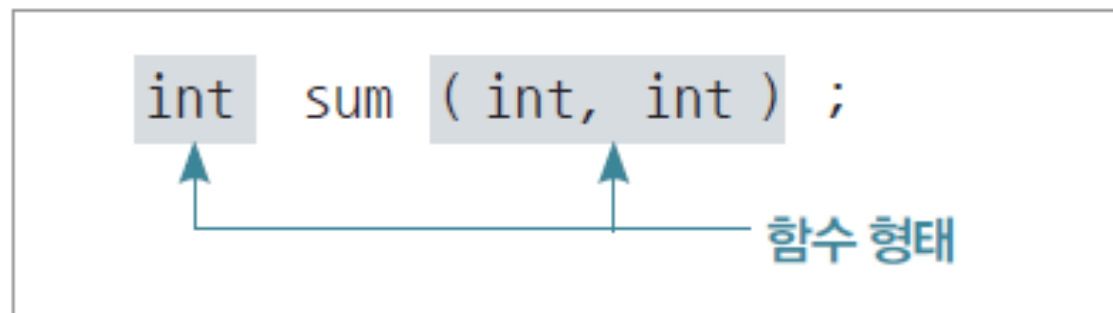
- ▶ 함수명이 주소라는 증거
 - ▶ 간접 참조 연산자를 사용한 결과
- ▶ 함수명에도 간접 참조 연산자 사용
 - ▶ 가리키는 함수 기능 사용 가능
 - ▶ sum 함수 다음과 같이 호출하는 것도 가능
- ▶ 함수의 주소도 포인터에 저장하면 포인터로 함수를 호출 가능
 - ▶ 주소 저장할 포인터
 - ▶ 주소가 가리키는 것과 동일 형태 가리키도록 선언
 - ▶ sum 함수의 형태 파악하는 것이 우선

```
(*sum)(10, 20); // 함수명에 괄호와 함께 간접참조 연산자를 사용하여 호출
```

함수 포인터의 개념

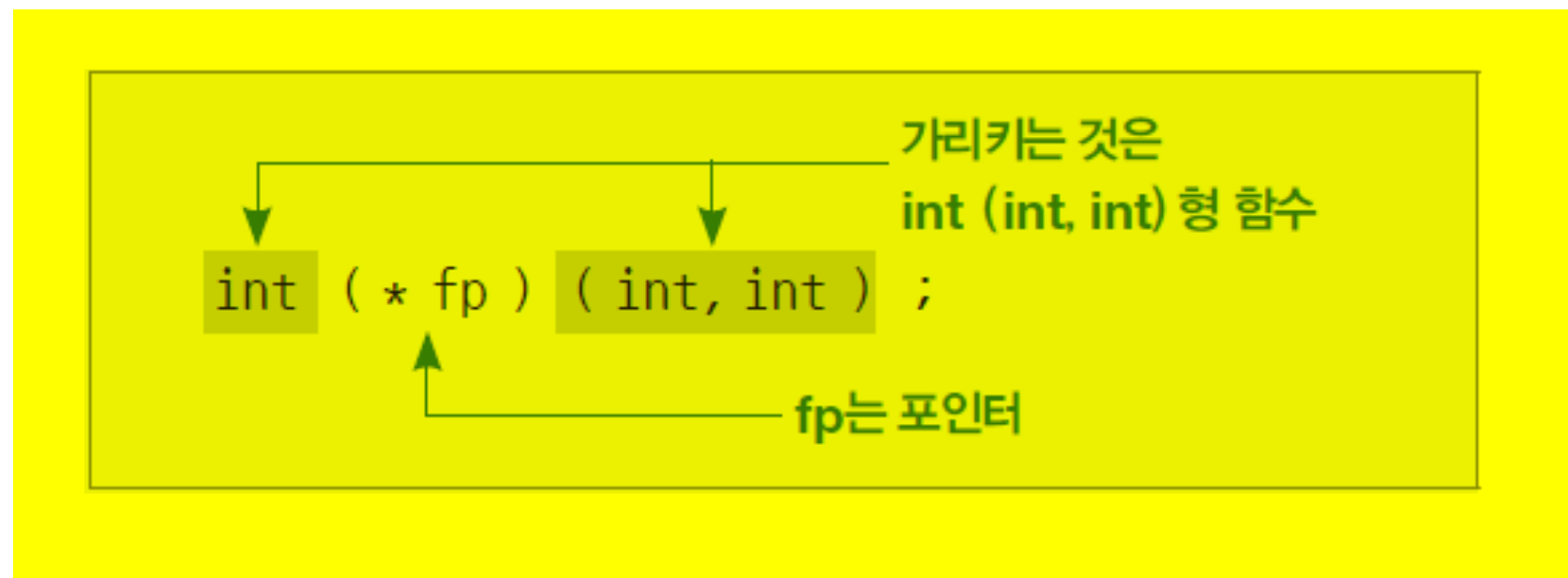
▶ 함수 형태

- ▶ 매개변수의 개수와 자료형, 반환값의 자료형으로 정의
- ▶ 함수 선언에서 알 수 있음
- ▶ 함수 포인터가 가리키는 형태



함수 포인터의 개념

- ▶ 함수의 주소 sum 저장할 함수 포인터 선언
 - ▶ 함수 포인터는 변수명 앞에 별(*)을 붙여 포인터 표시
 - ▶ 가리키는 함수의 형태를 반환값과 매개변수로 나누어 적음



함수 포인터의 개념

- ▶ 변수명을 별표와 함께 괄호로 묶어야
 - ▶ 괄호가 없으면 주소 반환하는 함수 선언 되므로 주의

```
int * fp (int, int) ;    // 두 정수를 인수로 받고 주소를 반환하는 함수의 선언
```

- ▶ 함수 포인터 선언 후 함수명 저장
 - ▶ 포인터를 함수처럼 사용 가능
 - ▶ 함수 포인터도 간접 참조 연산자 없이 함수 호출 가능

```
fp ( 10, 20 ) ;          // 함수 포인터로 함수 호출, (*fp)(10, 20)도 사용 가능
```

함수 포인터의 개념

- ▶ 함수 포인터 - 함수 형태만 같으면 모든 함수 사용 가능
 - ▶ 형태가 같은 다양한 기능의 함수 선택적으로 호출 가능

예제 15-8 함수 포인터로 원하는 함수를 호출하는 프로그램

```
1. #include <stdio.h>
2.
3. void func(int (*fp)(int, int));    // 함수 포인터를 매개변수로 갖는 함수
4. int sum(int a, int b);            // 두 정수를 더하는 함수
5. int mul(int a, int b);            // 두 정수를 곱하는 함수
6. int max(int a, int b);            // 두 정수 중에 큰 값을 구하는 함수
7.
8. int main(void)
9. {
10.     int sel;                       // 선택된 메뉴 번호를 저장할 변수
11.
```

함수 포인터의 개념

```
12.    printf("1. 두 정수의 합\n");        // 메뉴 출력
13.    printf("2. 두 정수의 곱\n");
14.    printf("3. 두 정수 중에서 큰 값 계산\n");
15.    printf("원하는 작업을 선택하세요 : ");
16.    scanf("%d", &sel);                // 메뉴 번호 입력
17.
18.    switch(sel)
19.    {
20.        case 1: func(sum); break;        // 1이면 func에 덧셈 기능 추가
21.        case 2: func(mul); break;        // 2이면 func에 곱셈 기능 추가
22.        case 3: func(max); break;        // 3이면 func에 큰 값 구하는 기능 추가
23.    }
24.
25.    return 0;
26. }
27.
```

함수 포인터의 개념

```
28. void func(int (*fp)(int, int))
29. {
30.     int a, b;                // 두 정수를 저장할 변수
31.     int res;                 // 함수의 반환값을 저장할 변수
32.
33.     printf("두 정수값을 입력하세요 : ");
34.     scanf("%d%d", &a, &b);    // 두 정수 입력
35.     res = fp(a, b);          // 함수 포인터로 가리키는 함수를 호출
36.     printf("결과값은 : %d\n", res); // 반환값 출력
37. }
38.
39. int sum(int a, int b)        // 덧셈 함수
40. {
41.     return (a + b);
42. }
43.
```

함수 포인터의 개념

```
44. int mul(int a, int b)                // 곱셈 함수
45. {
46.     return (a * b);
47. }
48.
49. int max(int a, int b)                // 큰 값을 구하는 함수
50. {
51.     if(a > b) return a;
52.     else return b;
53. }
```

실행
결과

```
1. 두 정수의 합
2. 두 정수의 곱
3. 두 정수 중에서 큰 값 계산
원하는 작업을 선택하세요 : 2 ☐
두 정수값을 입력하세요 : 3 7 ☐
결과값은 : 21
```

함수 포인터의 개념

- ▶ 함수 정의할 때 일부 구현하지 않고 함수 호출될 때 기능 결정
- ▶ 28행 func 함수
 - ▶ 2개의 정수를 키보드로부터 입력
 - ▶ 두 정수로 연산 수행
 - ▶ 연산 결과를 화면에 출력
 - ▶ 2번의 연산 종류 함수 호출할 때 결정
 - ▶ 매개변수에 함수 포인터 선언
 - ▶ func 함수를 호출할 때는 원하는 기능의 함수 인수로 주고 호출



함수 포인터의 활용

- ▶ func 함수 안에서는 함수 포인터인 매개변수 fp가 함수명을 저장하여 해당 함수를 가리킴
 - ▶ fp를 통해 해당 기능 가진 함수 호출

```
void func ( int (*fp)(int, int) )  
{  
    ...  
    fp(a, b); // 입력한 두 정수를 주고 fp가 가리키는 함수 호출  
    ...  
}
```

함수 포인터의 활용

- ▶ func 함수는 함수의 주소 받아 필요한 함수 호출
 - ▶ 필요한 함수 직접 호출해 같은 기능 하는 코드 가능
- ▶ 함수 포인터 써야 하는 경우
 - ▶ func 함수만 따로 만드는 경우
 - ▶ 만드는 시점에서 연산 방법을 결정할 수 없다면 일단 함수 포인터 쓰고 나중에 func 함수를 호출하는 곳에서 연산 방법 함수로 구현
 - ▶ 하나의 프로그램이 여러 개의 파일로 분리되어 있는 경우 다른 파일에 있는 정적 함수(static function) 호출하는 방법

VOID 포인터

- ▶ 주소는 가리키는 자료형이 일치하는 포인터에만 대입 가능
 - ▶ 가리키는 자료형이 다른 주소를 저장하는 경우
 - ▶ void 포인터 사용
 - ▶ Void 포인터란 가리키는 자료형 정해지지 않은 포인터

VOID 포인터

예제 15-9 void 포인터의 사용

```
1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     int a = 10;           // int형 변수
6.     double b = 3.5;       // double형 변수
7.     void *vp;             // void 포인터
8.
9.     vp = &a;              // int형 변수의 주소 저장
10.    printf("a : %d\n", *(int *)vp);
11.
12.    vp = &b;               // double형 변수의 주소 저장
13.    printf("b : %.1lf\n", *(double *)vp);
14.
15.    return 0;
16. }
```

실행
결과
a : 10
b : 3.5

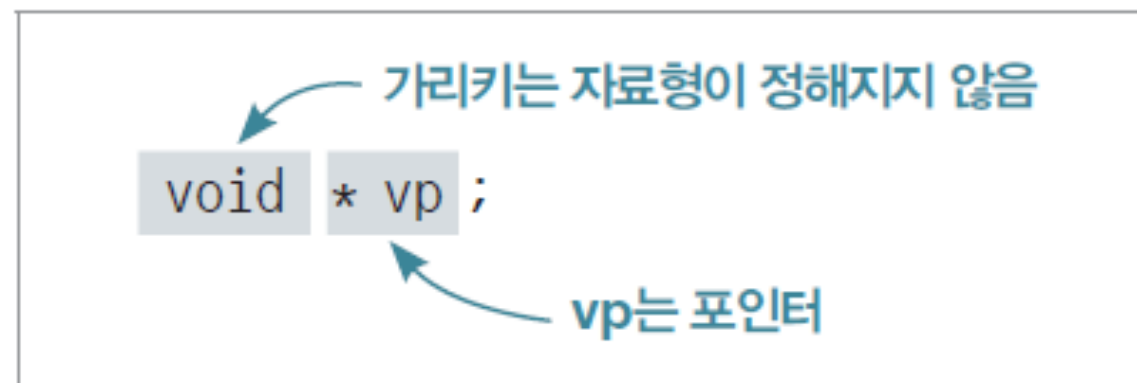
VOID 포인터

- ▶ void 포인터 선언 방법

- ▶ 변수명 앞에 별 (*) 붙여 포인터임을 표시

- ▶ 맨 앞에 void 적음

- ▶ void는 가리키는 자료형을 결정하지 않겠다는 뜻



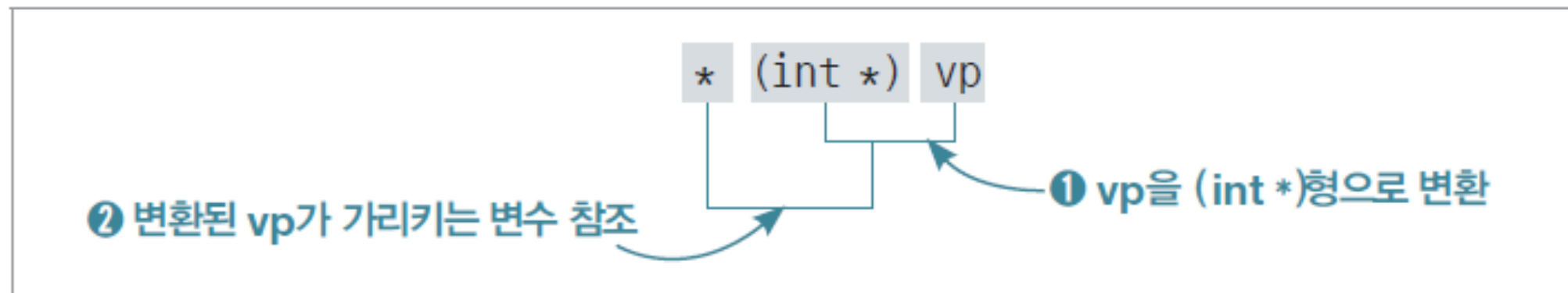
VOID 포인터

- ▶ 가리키는 자료형이 정해져 있지 않으므로 모든 주소 저장
- ▶ 간접참조 연산이나 정수 연산 불가능
- ▶ 간접참조 연산의 경우는 메모리의 특정 번지로 가서 몇 바이트를 어떤 형태로 읽어야 할지 알 수 없음
- ▶ 정수 연산도 얼마 곱해서 더해야 하는지 알 수 없음
- ▶ void 포인터 사용할 때는 원하는 형태로 변환하여 사용

```
printf("%d", *(int *)vp); // 형변환 후에 간접참조 연산자로 저장된 값 출력
(int *)vp+1;             // 형변환 후에 정수를 더한다.
```

VOID 포인터

- ▶ 원래의 자료형에 맞게 void 포인터 형변환
 - ▶ 형변환 후 각각 가리키는 변수 출력 위해 간접참조 연산
- ▶ 형변환 연산자와 간접참조 연산자는 모두 단항 연산자
 - ▶ 우선순위 같으므로 오른쪽-> 왼쪽으로 차례로 연산



VOID 포인터

- ▶ 대입 연산 할 때
 - ▶ 형변환 없이 void 포인터를 다른 포인터에 대입 가능
 - ▶ 항상 명시적으로 형변환 하여 사용할 것
 - ▶ Ex) int *pi를 double *pi 로 잘못 작성
 - ▶ 컴파일 오류가 발생

```
int *pi = (int *)vp;
```

```
// 명시적으로 형변환하여 대입하는 것이 좋다!
```
