8장 포인터

김명호

내용

- 포인터 소개
- 주소연산자 &
- 포인터 변수
- 역참조 연산자 *
- void 포인터
- 포인터 연산
- 함수와 포인터
- 메모리 사상함수
- 동적메모리 할당
- 포인터 배열
- const, restrict
- 함수 포인터

포인터

- 지금까지 할당 받은 메모리 공간은 변수 이름으로 접근했었음
  - 예

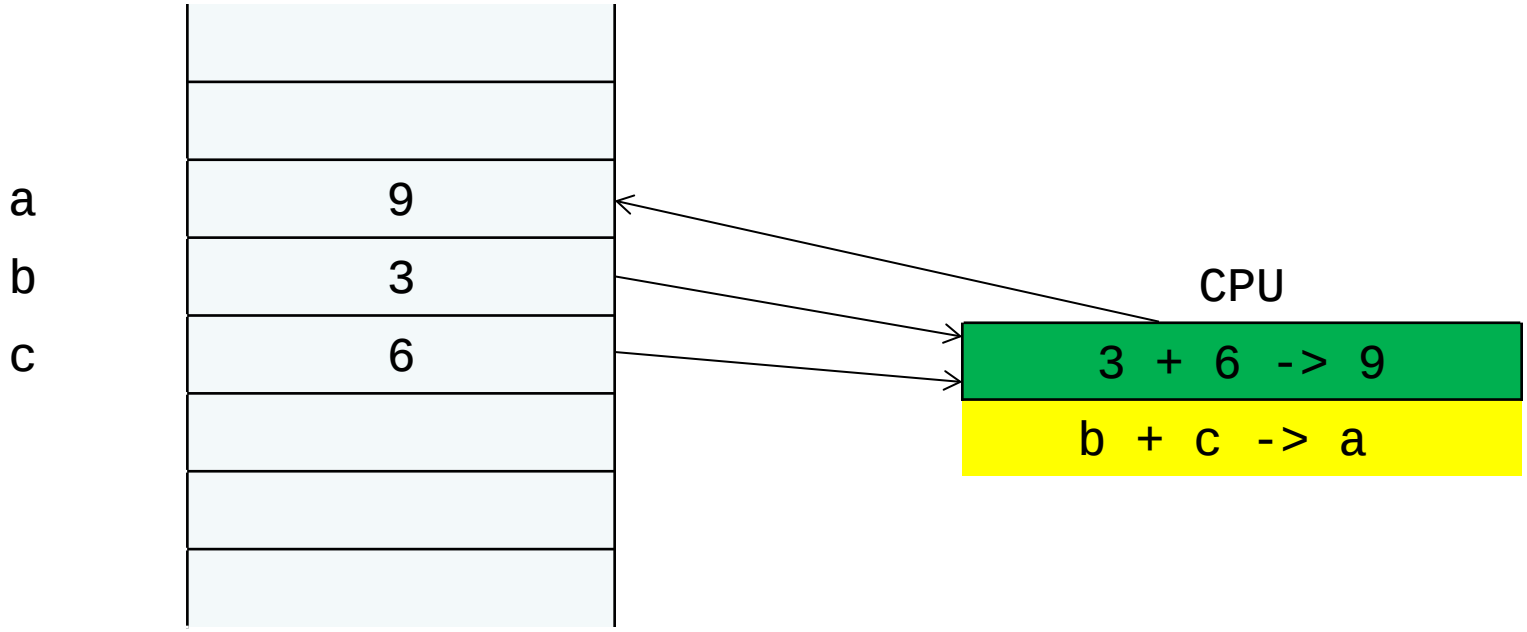
```
int a, b, c;    // a, b, c를 위한 메모리 할당
a = b + c;      // a, b, c 이름으로 메모리 접근
```

- 포인터는 데이터를 가진 메모리 공간을 주소로 접근하기 위해 사용
- 메모리는 각 바이트 별로 주소가 붙여진 1 차원 배열

변수 사용

- 예

```
int a, b = 3, c = 6;
a = b + c;
```

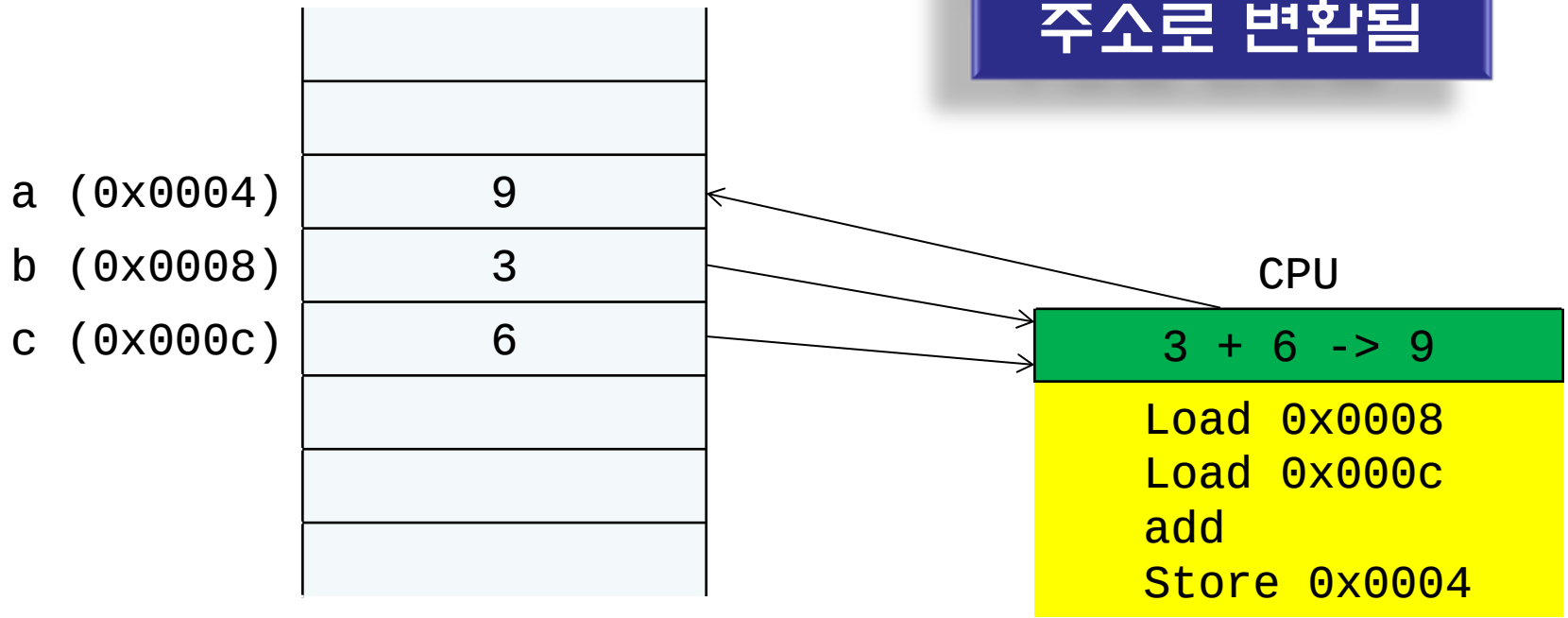


변수 사용

- 예

```
int a, b = 3, c = 6;
a = b + c;
```

모든 변수는 컴파일 후 할당 받은 주소로 변환됨



주소연산자 &

- 주소 연산자 &는 변수 앞에 오고, 주소 연산 수식의 값은 그 변수가 할당 받은 메모리 주소임
  - 예

```
int  i;    // i를 위한 메모리가 할당됨
&i         // i를 위해 할당된 메모리 주소
```

- 상수나 수식 앞에는 주소 연산자 &를 사용할 수 없음
  - 잘못 사용한 예

```
&3          // 상수 앞에서 사용
&(i + 3)    // 수식 앞에서 사용
```

포인터 변수

- **값으로 주소를 갖는 변수**
  - 주소 연산자 &와 함께 많이 사용됨
- **선언 방법**
  - 일반 변수와 유사하게 선언하고, 변수 이름 앞에 *를 붙여 명시함
  - 예

```
int *p;       // int 형 포인터 변수 p 선언
int *a, b;    // 포인터 a와 int 변수 b 선언
int *x, *y;   // 포인터 x와 y 선언
```

포인터 변수

- 포인터 변수가 가질 수 있는 값의 범위
  - 특정주소 0
  - 주어진 C 시스템에서 기계 주소로 해석될 수 있는 양의 정수 집합

포인터 변수

- 사용 예제

```
int i, *p;
register int v;

p = 0;
p = &i;
p = (int *) 1776;
p = 3000;
p = &(i + 99);
p = &v;
```

포인터 변수

- **사용 예제**

```
int i, *p;
register int v;

p = 0;              // p = NULL;
p = &i;
p = (int *) 1776; // int 형 상수를 (int *)로 캐스트
p = 3000;           // 오류, 상수에 사용
p = &(i + 99);      // 오류, 수식에 사용
p = &v;             // 오류, register 변수에 사용
```

포인터 변수

- **포인터 변수 사용**

```
int i = 100, *p;
p = &i;
```

- p가 i의 주소를 가짐
- p가 i를 포인트 함

역참조 연산자 *

- 간접지정 연산자라고도 함
- 단항 연산자, 우에서 좌로의 결합 순위
- p가 포인터라면, *p는 p가 주소인 변수의 값을 나타냄

  - p : i의 주소
  - *p (= i) : 100

예제 프로그램

프로그램 8.1

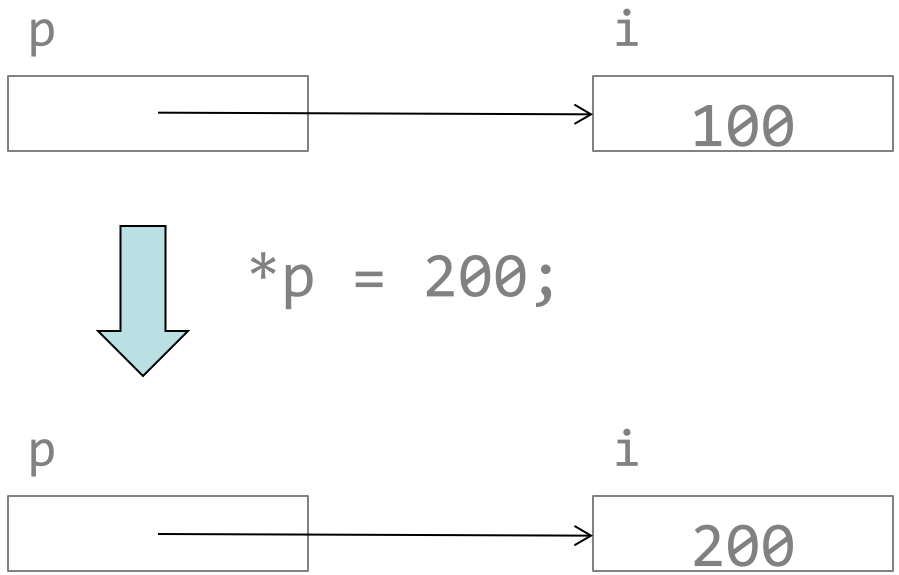
```
#include <stdio.h>
int main(void){
   int i = 100;
   int *p;
   p = &i;
   printf("i 주소 : %p\n", &i);
   printf(" i 값  : %d\n", i);
   printf(" p 값  : %p\n", p);
   printf("*p 값  : %d\n", *p);
   return 0;
}
```

프로그램 결과

```
i 주소 : 0x23efc4  // 시스템 마다 다름
 i 값  : 100
 p 값  : 0x23efc4  // 시스템 마다 다름
*p 값  : 100
```

역참조 연산자 *

- 역참조 연산자가 붙은 포인터는 일반 변수와 같이 배정 연산자 좌측에 올 수도 있음
  - 그 포인터가 포인트하고 있는 위치를 나타냄



예제 프로그램

프로그램 8.2

```
#include <stdio.h>
int main(void){
   int i, j = 5, *p;
   p = &i;
   i = 10;
   printf("(p = &i)i = %d, j = %d, *p = %d\n", i, j, *p);
   *p = *p * j;
   printf("(p = &i)i = %d, j = %d, *p = %d\n", i, j, *p);
   p = &j;
   printf("(p = &j)i = %d, j = %d, *p = %d\n", i, j, *p);
   return 0;
}
```

프로그램 결과

```
(p = &i)i = 10, j = 5, *p = 10
(p = &i)i = 50, j = 5, *p = 50
(p = &j)i = 50, j = 5, *p = 5
```

void 포인터 변수

- void 형 포인터 변수
  - 형이 없는 포인터

```
int *p;
float x;
void *v;
p = &x;        // 오류, 두 형이 다름
v = &x;
p = v;
```

void 포인터 변수

- void 형 포인터 변수를 사용할 때에는 적절한 형 변환 필요

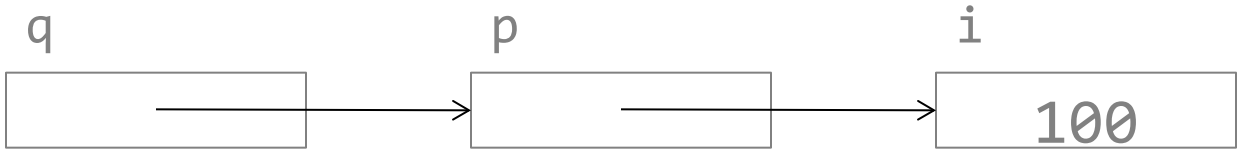
```
int *p, i, j = 20;
void *v;
v = &j;
i = *((int *)v) + 10;
// v가 포인트 하는 곳의 데이터 형을 int 형으로 다룸
```

포인터의 포인터 변수

- 포인터 변수를 포인트 하는 포인터

```
int i = 100, *p = &i, **q = &p;
```

  - q는 포인터의 포인터 변수
  - 선언문에서 **로 명시



  - *q : p
  - **q : i

예제 프로그램

프로그램 8.3

```
#include <stdio.h>
int main(void){
   int i = 100, j = 200, *p = &i, **q = &p;
   printf("i = %d, j = %d, *p = %d, **q = %d\n", i, j, *p, **q);
   printf("&i = %p, &j = %p, p = %p, *q = %p\n", &i, &j, p, *q);
   printf("&p = %p, q = %p\n", &p, q);
   *q = &j;    // p = &j
   printf("i = %d, j = %d, *p = %d, **q = %d\n", i, j, *p, **q);
   printf("&i = %p, &j = %p, p = %p, *q = %p\n", &i, &j, p, *q);
   printf("&p = %p, q = %p\n", &p, q);
   return 0;
}
```

프로그램 결과

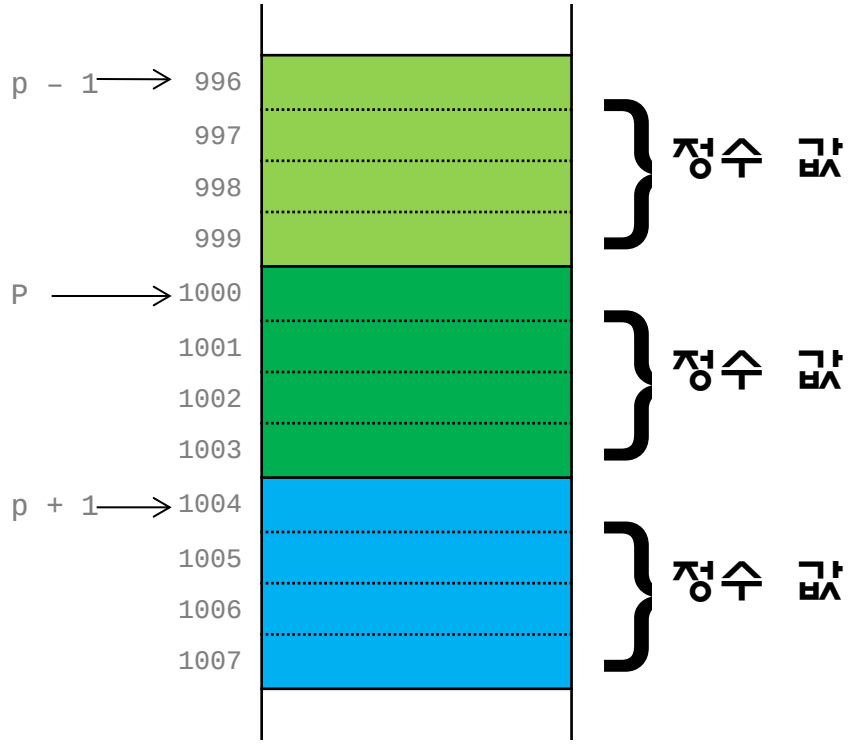
```
i = 100, j = 200, *p = 100, **q = 100
&i = 0x23efc4, &j = 0x23efc0, p = 0x23efc4, *q = 0x23efc4
&p = 0x23efbc, q = 0x23efbc
i = 100, j = 200, *p = 200, **q = 200
&i = 0x23efc4, &j = 0x23efc0, p = 0x23efc0, *q = 0x23efc0
&p = 0x23efbc, q = 0x23efbc
```

포인터 연산

- 포인터 연산은 C의 강력한 특징 중 하나
- p와 q가 포인터 변수라면
  - p + i, p - i : p가 포인트하고 있는 곳으로부터 i번째 앞 또는 뒤 원소
  - p - q : p와 q 사이의 원소 개수

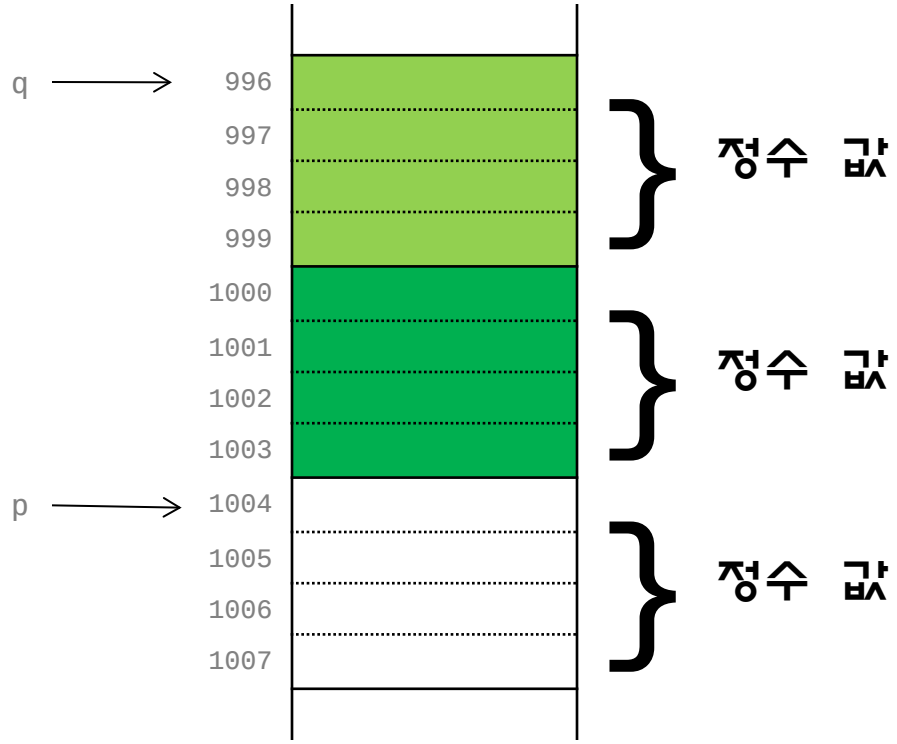
포인터 연산

- p, p +1, p -1
  - 가정 : p가 int 형 포인터, 1000 번지를 포인트



포인터 연산

- p - q
  - 가정 : p와 q가 int 형 포인터



$\therefore p - q : 2$

예제 프로그램

프로그램 8.4

```
#include <stdio.h>
int main(void){
   double x[10], *p, *q;

   p = &x[2];
   q = p + 5;
   printf("q - p = %d\n", q - p);
   printf("(int) q - (int) p = %d\n", (int) q - (int) p);
   return 0;
}
```

프로그램 결과

```
q - p = 5

(int) q - (int) p = 40
```

포인터와 함수

- C는 기본적으로 "값에 의한 호출" 메커니즘 사용
- "참조에 의한 호출"의 효과를 얻기 위해서는 함수 정의의 매개변수 목록에서 포인터를 사용해야 함

값에 의한 호출

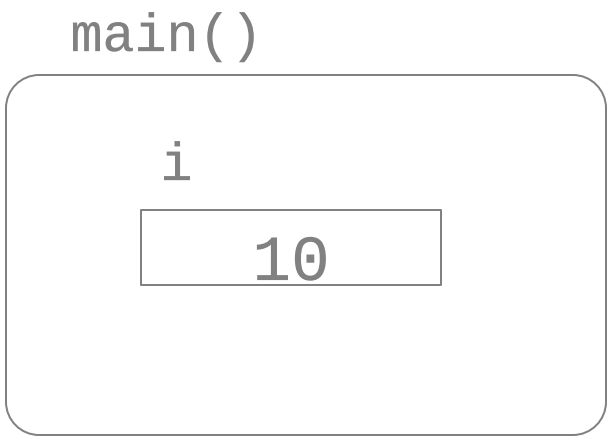
```
void change_it(int k);
int main(void){
   int i = 10;
   printf("함수 호출 이전의 i 값 : %d\n", i);
   change_it(i);
   printf("함수 호출 이후의 i 값 : %d\n", i);
   return 0;
}
```

```
void change_it(int k){
   k = k * k;
}
```

값에 의한 호출

```c
void change_it(int k);
int main(void){
   int i = 10;
   printf("함수 호출 이전의 i 값 : %d\n", i);
   change_it(i);
   printf("함수 호출 이후의 i 값 : %d\n", i);
   return 0;
}
```

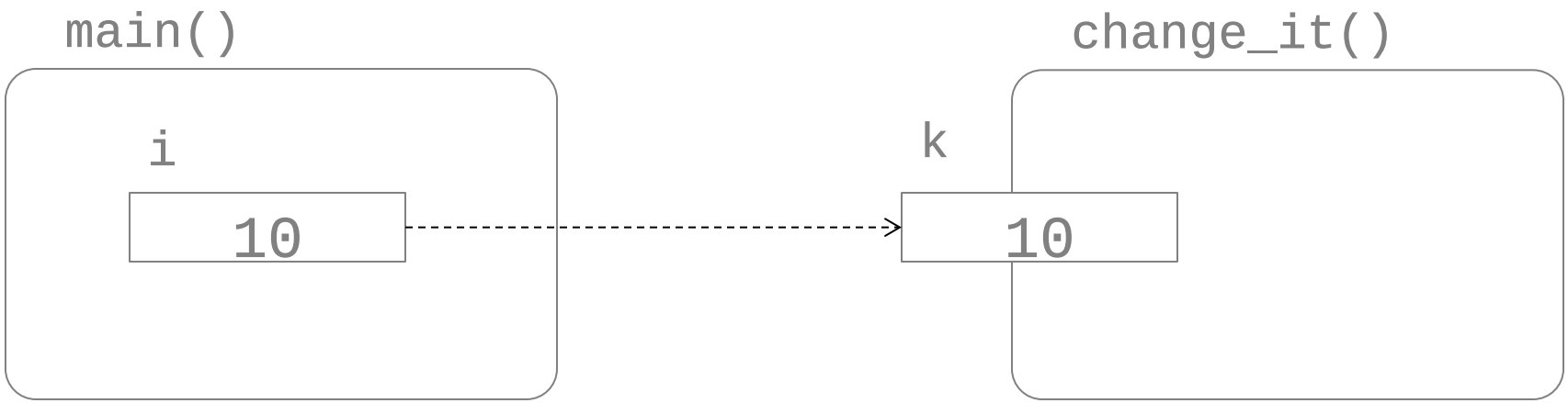
```c
void change_it(int k){
   k = k * k;
}
```



값에 의한 호출

```c
void change_it(int k);
int main(void){
    int i = 10;
    printf("함수 호출 이전의 i 값 : %d\n", i);
    change_it(i);
    printf("함수 호출 이후의 i 값 : %d\n", i);
    return 0;
}
```

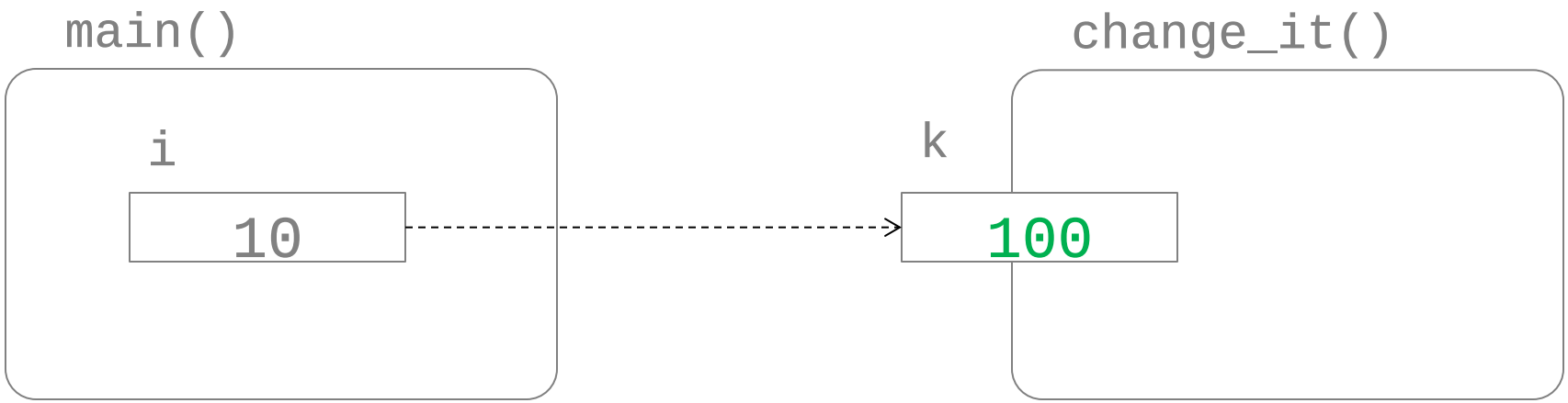
```c
void change_it(int k){
    k = k * k;
}
```



값에 의한 호출

```c
void change_it(int k);
int main(void){
    int i = 10;
    printf("함수 호출 이전의 i 값 : %d\n", i);
    change_it(i);
    printf("함수 호출 이후의 i 값 : %d\n", i);
    return 0;
}
```

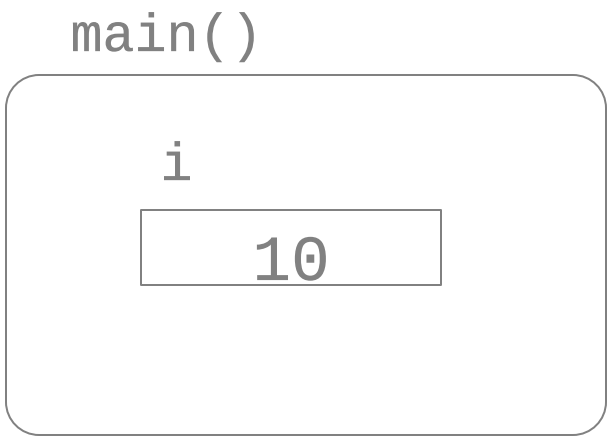
```c
void change_it(int k){
    k = k * k;
}
```



값에 의한 호출

```c
void change_it(int k);
int main(void){
   int i = 10;
   printf("함수 호출 이전의 i 값 : %d\n", i);
   change_it(i);
   printf("함수 호출 이후의 i 값 : %d\n", i);
   return 0;
}
```

```c
void change_it(int k){
   k = k * k;
}
```



값에 의한 호출

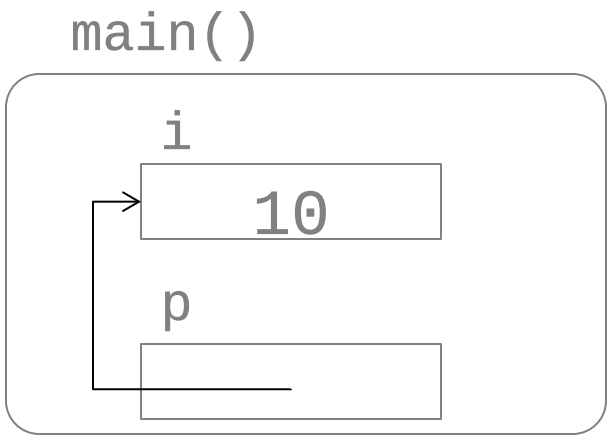
```
void change_it(int k);
int main(void){
   int i = 10;
   printf("함수 호출 이전의 i 값 : %d\n", i);
   change_it(i);
   printf("함수 호출 이후의 i 값 : %d\n", i);
   return 0;
}
```

```
void change_it(int k){
   k = k * k;
}
```

참조에 의한 호출

```
void change_it_p(int *k);
int main(void){
    int i = 10;
    int *p = &i;
    printf("함수 호출 이전의 i 값 : %d\n", i);
    change_it_p(p);
    printf("함수 호출 이후의 i 값 : %d\n", i);
    return 0;
}
```

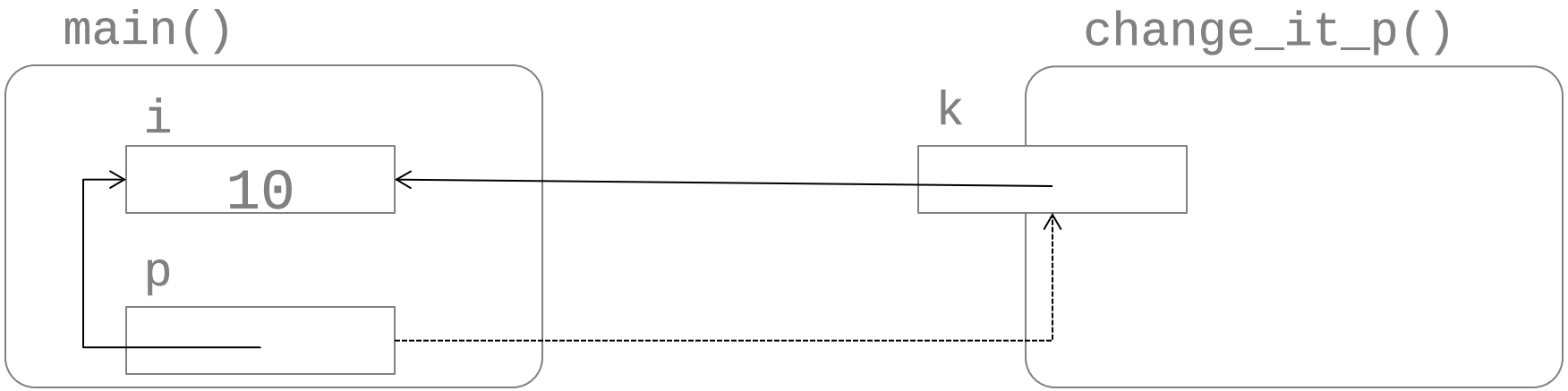
```
void change_it_p(int *k){
    *k = *k * *k;
}
```



참조에 의한 호출

```c
void change_it_p(int *k);
int main(void){
    int i = 10;
    int *p = &i;
    printf("함수 호출 이전의 i 값 : %d\n", i);
    change_it_p(p);
    printf("함수 호출 이후의 i 값 : %d\n", i);
    return 0;
}
```

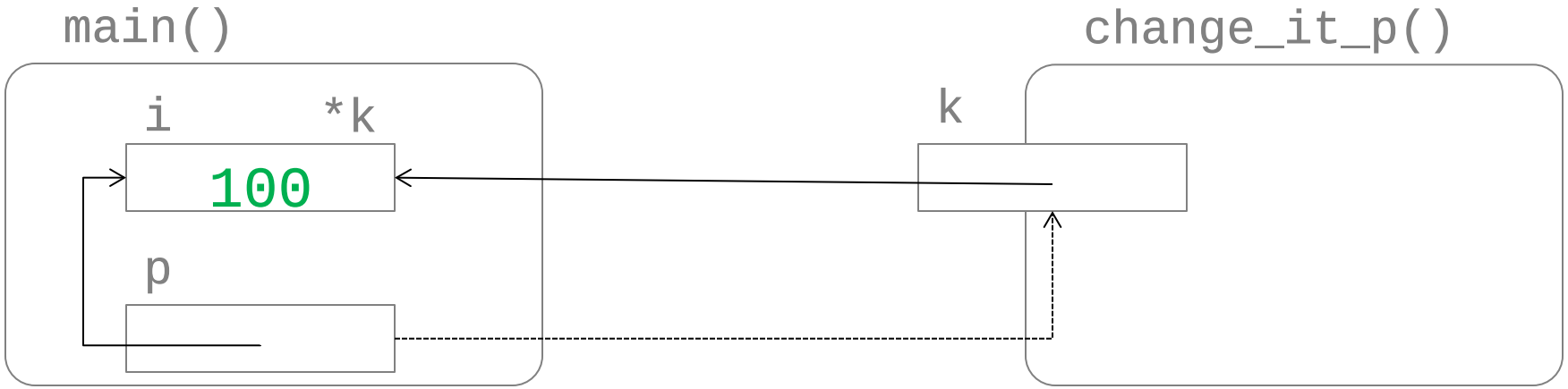
```c
void change_it_p(int *k){
    *k = *k * *k;
}
```



참조에 의한 호출

```c
void change_it_p(int *k);
int main(void){
    int i = 10;
    int *p = &i;
    printf("함수 호출 이전의 i 값 : %d\n", i);
    change_it_p(p);
    printf("함수 호출 이후의 i 값 : %d\n", i);
    return 0;
}
```

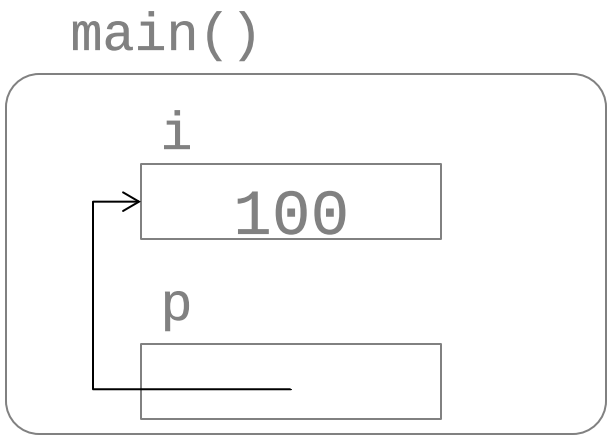
```c
void change_it_p(int *k){
    *k = *k * *k;
}
```



참조에 의한 호출

```
void change_it_p(int *k);
int main(void){
   int i = 10;
   int *p = &i;
   printf("함수 호출 이전의 i 값 : %d\n", i);
   change_it_p(p);
   printf("함수 호출 이후의 i 값 : %d\n", i);
   return 0;
}
```

```
void change_it_p(int *k){
   *k = *k * *k;
}
```



참조에 의한 호출

```
void change_it_p(int *k);
int main(void){
   int i = 10;
   int *p = &i;
   printf("함수 호출 이전의 i 값 : %d\n", i);
   change_it_p(p);
   printf("함수 호출 이후의 i 값 : %d\n", i);
   return 0;
}
```

```
void change_it_p(int *k){
   *k = *k * *k;
}
```

참조에 의한 호출

- "참조에 의한 호출"의 효과를 얻는 방법
  1. 함수 매개변수를 포인터형으로 선언
  2. 함수 몸체에서 역참조 포인터 사용
  3. 함수를 호출할 때 주소를 인자로 전달

여러 개 값 리턴

프로그램 8.7

```
int divide_p(int, int, int *, int *);
int main(void){
   int i, j, q, r;
   printf("피제수를 입력하세요 : ");
   scanf("%d", &i);
   printf("제수를 입력하세요 : ");
   scanf("%d", &j);
   if (divide_p(i, j, &q, &r))
      printf("0으로 나눌 수 없습니다.\n");
   else
      printf("%d / %d : 몫은 %d이고 나머지는 %d입니다.\n",
              i, j, q, r);
   return 0;
}
```

여러 개의 값 리턴

프로그램 8.7

```
int divide_p(int dividend, int divisor,
                           int * quotient, int * rem)
{
   if (divisor == 0)
      return -1;
   *quotient = dividend / divisor;
   *rem = dividend % divisor;
   return 0;
}
```

프로그램 결과

```
피제수를 입력하세요 : 541
제수를 입력하세요 : 99
541를(을) 99(으)로 나누면, 몫이 5이고 나머지는 46입니다.
```

여러 개의 값 리턴

```
int divide_p(int dividend, int divisor,
             int * quotient, int * rem)
```

- divide_p()는 호출한 곳으로 3개의 값을 전달함
  - quotient : 몫
  - rem : 나머지
  - 0 or -1 : 상태 값 (return을 통해)

scanf()

- scanf()에서 읽어드린 값을 저장할 인자 앞에 &를 사용
  - 예

```
scanf("%d", &i);
```

  - scanf() 함수에서 입력을 처리한 결과를 받기 위한 것임

포인터와 배열

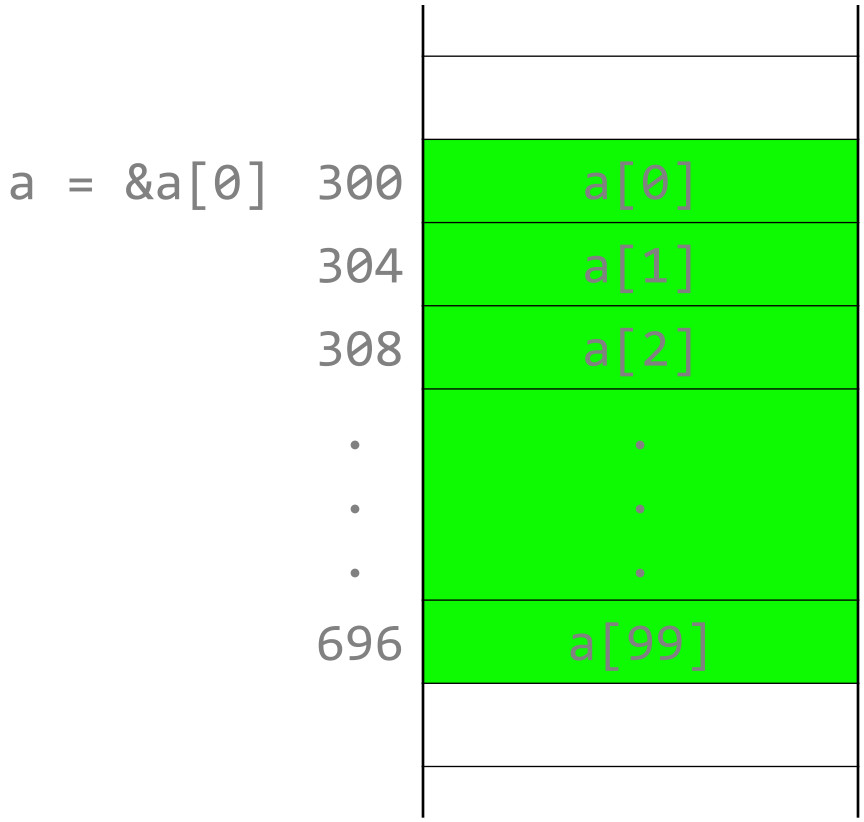
- 배열과 포인터는 밀접한 관련이 있음
- 배열 이름은 포인터이고, 그 값은 배열의 첫 번째 원소의 주소 값임
- 배열과 포인터에는 둘 다 배열의 원소를 지정하는 첨자를 사용할 수 있음
- 포인터 변수는 다른 주소들을 값으로 가질 수 있지만, 배열 이름은 고정된 주소를 갖는 상수 포인터임

배열과 포인터의 관계

```
#define   N   100
int  a[N], i, *p;  // a : int 형 포인터, a[0]의 주소
p = a;             // p = &a[0];
p = a + i;
    // a + i : a[0]에서 i번째 떨어진 원소 위치(주소)
*(a + i) = 10;
p[i] = 10;         // p[i] == *(p + i)
*(p + i) = 10;
a = &i;            // 오류
```

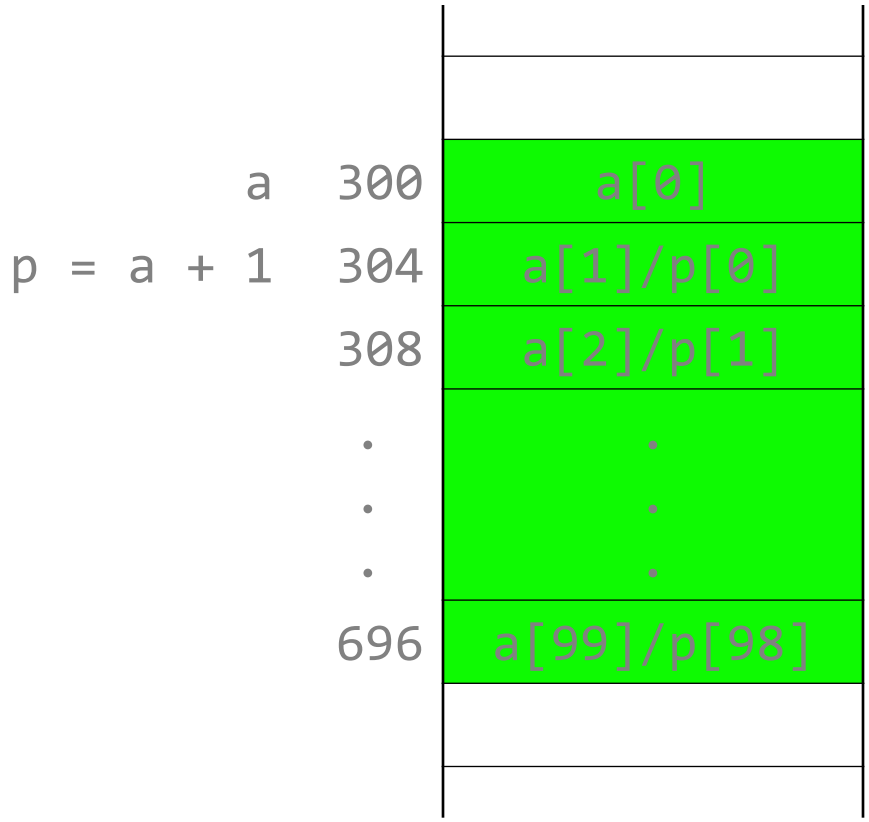
배열과 포인터의 관계

```
#define   N   100
int  a[N], i, *p;          // &a[0] : 300 번지
```



배열과 포인터의 관계

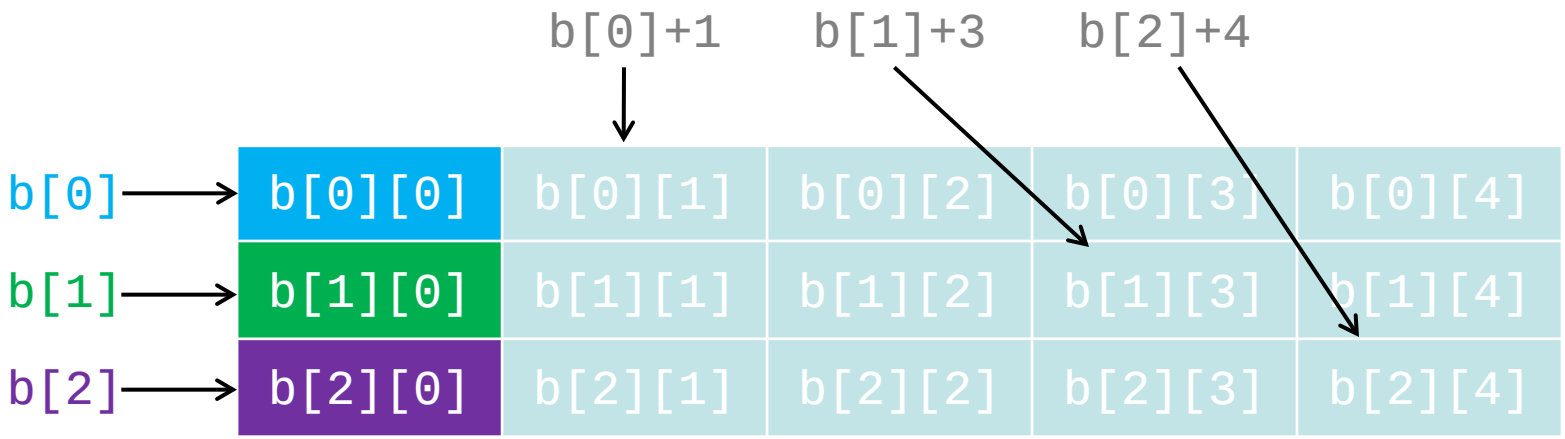
```
int  a[N], i, *p;          // &a[0] : 300 번지
p = a + 1;                 // p = &a[1];
```



포인터와 배열

- 2차원 배열도 인덱스를 하나 제거하면 포인터가 됨

```
int  b[3][5];
- b[i] : int 형 포인터, &b[i][0]
- b[i] + j : &b[i][j]
```

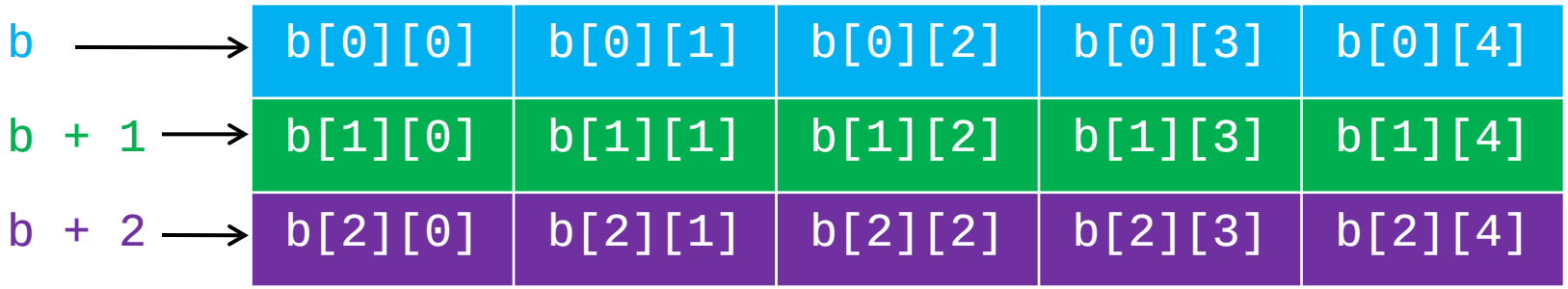


포인터와 배열

- 2차원 배열에서 인덱스를 두 개 제거하면 포인터의 포인터가 됨

```
int  b[3][5];
```

- b : int [5] 형 포인터
- b + i : &b[i][0]부터 int형 원소 5개 포인트



포인터와 배열

- **이차원 배열 원소인 b[i][j]의 다양한 표기 방법**

```
#define   N   100
int  b[N][N];
```

  - b[i][j]
  - *(b[i] + j)
  - *(*(b + i) + j)
  - (*(b + i))[j]

배열과 함수 보충

- **배열을 매개변수로 갖는 함수**
  - 배열 매개변수는 포인터임

```
int grade_sum2(int gr[], int size){
    int sum, i;
    for (sum = 0, i = 0; i < size; i++)
        sum += gr[i];
    return sum;
}
```

- gr : **포인터**

배열과 함수 보충

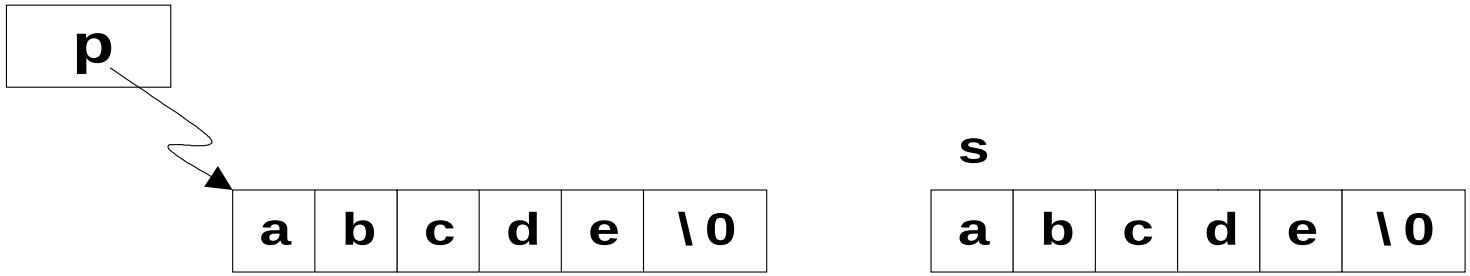
- **배열을 매개변수로 갖는 함수 호출**
  - 대응 인자는 주소 값이면 됨

```
#define N 100
int i, a[N], sum;
. . .
sum = grade_sum2(a, N);
      // sum = a[0] + a[1] + ... + a[99]
sum = grade_sum2(&a[5], 10);
      // sum = a[5] + a[6] + ... + a[14]
sum = grade_sum2(&i, 1);
      // sum = i
```

문자열과 포인터

- **문자 배열과 문자열 포인터 차이**

```
char *p = "abcde";
char s[] = "abcde";
```



문자열과 포인터

- **문자 배열과 문자열 포인터 차이**

```
char *p = "abcde";
char s[] = "abcde";

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

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

문자열과 포인터

- **문자 배열과 문자열 포인터 차이**

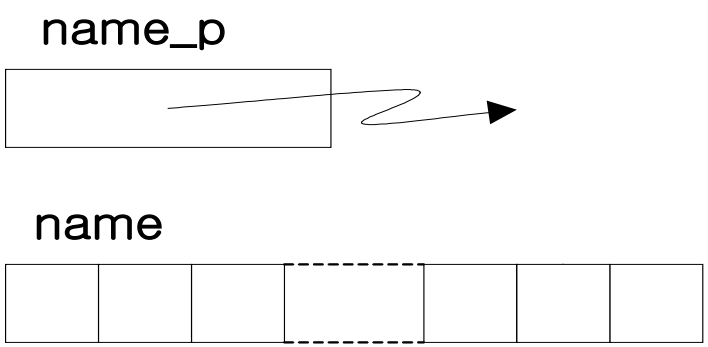
```
char *p = "abcde";
char s[] = "abcde";

p[i] = ‘x’;
p = s + 3;
s = p;
```

문자열과 포인터

- **문자 배열과 문자열 포인터 차이**

```
#define  N  20
char name[N] = "";
char *name_p;
scanf("%s", name);
scanf("%s", name_p);        // 오류 ??
```



메모리 사상 함수

- 배열 표기법은 컴파일 후 포인터 수식으로 변환
- 1차원 배열

```
int  a[5];
```

  - 프로그램 코드에서 a[i]는 컴파일 후 *(&a[0]+i)가 됨
    - &a[0] : a[0] 원소의 주소
    - &a[0] + i : a[0]으로부터 i번째 원소의 주소
    - *(&a[0] + i) : a[0]으로부터 i번째 원소

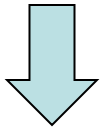
메모리 사상 함수

- **2차원 배열**

```
int  b[3][5];
```

  - 2차원 배열은 1차원인 메모리에 행우선으로 저장

b[0][0]	b[0][1]	b[0][2]	b[0][3]	b[0][4]
b[1][0]	b[1][1]	b[1][2]	b[1][3]	b[1][4]
b[2][0]	b[2][1]	b[2][2]	b[2][3]	b[2][4]



b[0][0]	b[0][1]	b[0][2]	b[0][3]	b[0][4]	b[1][0]	b[1][1]	b[1][2]	b[1][3]	b[1][4]	b[2][0]	b[2][1]	b[2][2]	b[2][3]	b[2][4]

메모리 사상 함수

- **2차원 배열**

```
int  b[3][5];
```

b[0][0]	b[0][1]	b[0][2]	b[0][3]	b[0][4]	b[1][0]	b[1][1]	b[1][2]	b[1][3]	b[1][4]	b[2][0]	b[2][1]	b[2][2]	b[2][3]	b[2][4]

- b[1][4]는 b[0][0]로부터 5 * 1 + 4 번째 원소
- b[2][3]은 b[0][0]로부터 5 * 2 + 3 번째 원소

→ b[i][j] : *(&b[0][0] + 5 * i + j)

메모리 사상 함수

- 3차원 배열

```
int  c[2][3][5];
```

  - `c[i][j][k] : *(&c[0][0][0] + 3*5*i + 5*j + k)`

- 다차원 배열
  - 같은 원리가 적용됨
  - 배열 차원이 높아질 수록 원소를 접근하기 위해 추가적으로 필요한 * 연산과 + 연산이 많아짐

메모리 사상 함수

- 함수 헤더에서 배열 매개변수의 가장 상위 차원은 명시하지 않아도 됨

```
int grade_sum_id(int gr[][M], int id)
```

  - 메모리 사상 함수에서 가장 상위 차원의 크기는 중요하지 않기 때문 임

메모리 사상 함수

- 다차원 배열을 사용할 때 주의점
  - 배열을 사용하면 프로그래밍은 쉬워지지만 추가적인 *와 + 연산이 요구됨
  - 프로그램의 속도가 떨어질 수 있음
  - 포인터를 사용하면 프로그램 속도를 높일 수 있음

동적 메모리 할당

- 효율적인 메모리 사용을 위해 함
- 메모리 할당 함수 : calloc(), malloc()

```
<stdlib.h>
void *calloc(size_t N, size_t el_size);
void *malloc(size_t N_bytes);
```

  - 두 함수 모두 할당된 메모리 주소를 리턴함
- 프로그래머는 calloc()과 malloc()을 사용하여 배열, 구조체, 공용체를 위한 공간을 동적으로 생성함
- 사용 후에는 free()를 사용하여 반납

동적 메모리 할당

프로그램 8.8 일부

```
. . .

. . .
for (i = 0; i < N; i++)
   scanf("%d", &grade[i]);
. . .
```

동적 메모리 할당

프로그램 8.8 일부

```
#include <stdlib.h>
. . .
int *grade, N;
. . .
scanf("%d", &N);
. . .
grade = (int *)calloc(N, sizeof(int));
// 또는  grade = (int *)malloc(N * sizeof(int));
. . .
for (i = 0; i < N; i++)
   scanf("%d", &grade[i]);
. . .
free(grade);
```

동적 메모리 할당

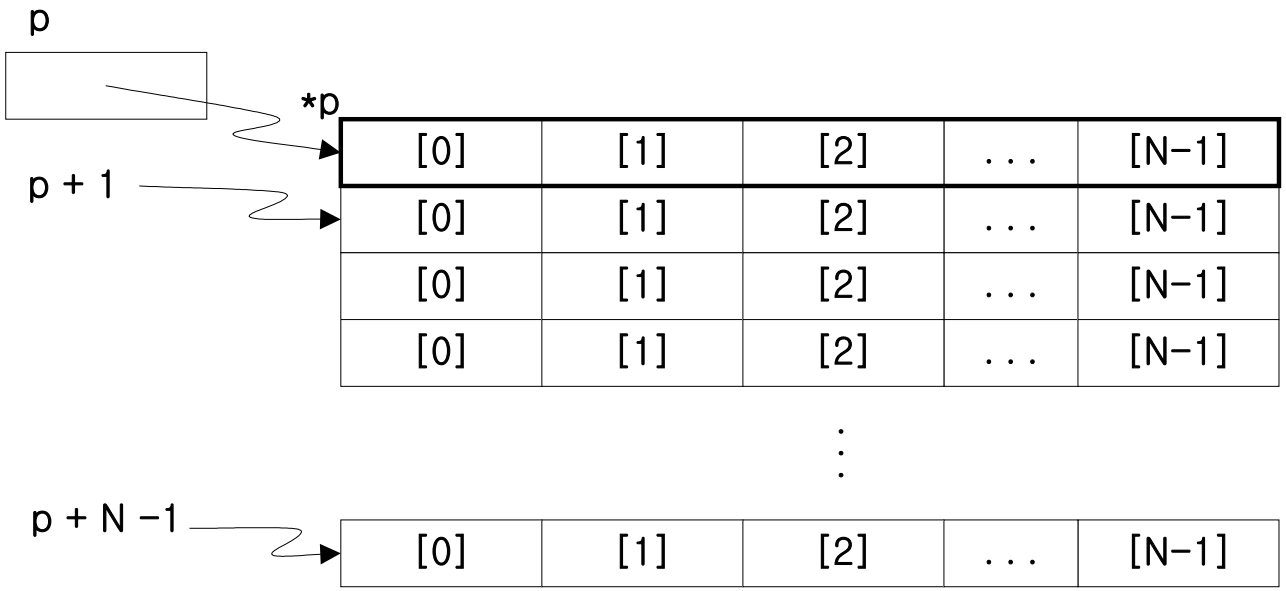
프로그램 8.9 일부

```
int (*p)[N], *q;        // p : 2차원 배열을 포인트 하기 위해
. . .
p = (int (*)[N])calloc(N * N, sizeof(int));
q = (int *) p;
for (i = 0; i < N * N; i++)
   q[i] = i;
for (i = 0; i < N; i++){
   for (j = 0; j < N; j++)
      printf("%3d ", p[i][j]);
   putchar('\n');
}
free(p);
```

동적 메모리 할당

- **예제 코드**

```
int (*p)[N], *q;          // p : 2차원 배열을 포인트 하기 위해
. . .
p = (int (*)[N])calloc(N * N, sizeof(int));
q = (int *) p;
```



프로그램 결과

```
 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

포인터 배열

- 단어 정렬 프로그램(프로그램 7.12)에서…

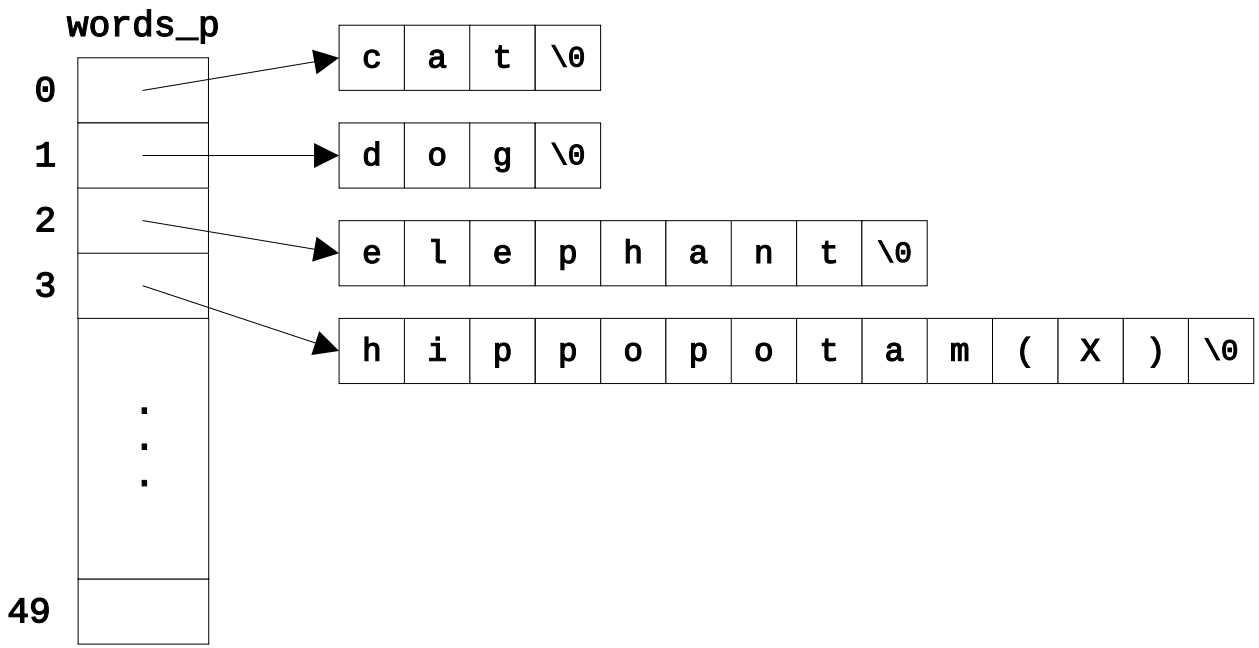
```
char words[N][M];      // N = 50, M = 14
```

words	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
words[0]	c	a	t	\0										
words[1]	d	o	g	\0										
words[2]	e	l	e	p	h	a	n	t	\0					
words[3]	h	i	p	p	o	p	o	t	a	m	(	X	)	\0
words[4]	s	e	a		h	o	r	s	e	(	X	)	\0	
words[5]	w	h	a	l	e	\0								
.														
.														
words[49]	.	.	.											

* 회색 부분 : 메모리 낭비

포인터 배열

- 다음과 같이 저장하면 메모리를 효율적으로 사용할 수 있음




```
char * words_p[N];
```

포인터 배열

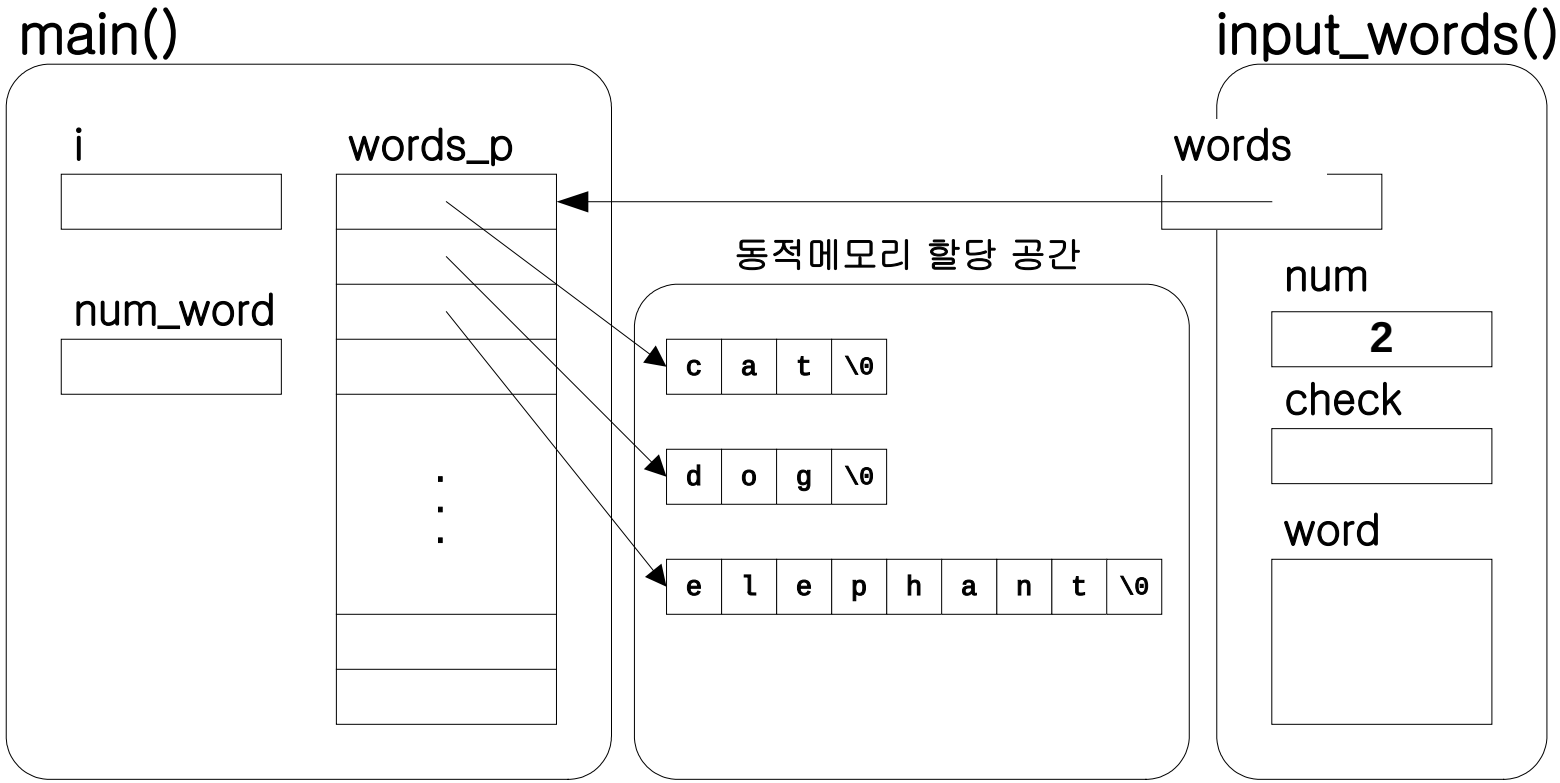
프로그램 8.10 일부

```
main(void):
    char *words_p[N];     // char * 형 배열
    input_words(words_p);
    for (i = 0; i < num_word; i++)
        free(words_p[i]);

input_words(char *words[]):
    char word[11];
    input_a_word(word);
    words[num] = (char *)calloc(strlen(word)+1, sizeof(char));
    strcpy(words[num], word);
```

포인터 배열

- **프로그램 8.10**



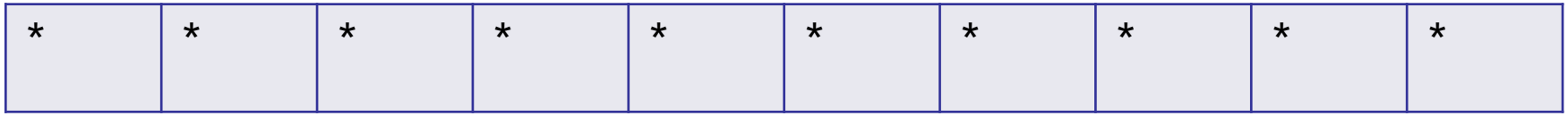
비교

- `double x;`
- `double *p[10];`
- `double (*q)[10];`

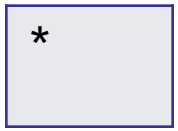
비교

- `double x;`

- `double *p[10];`



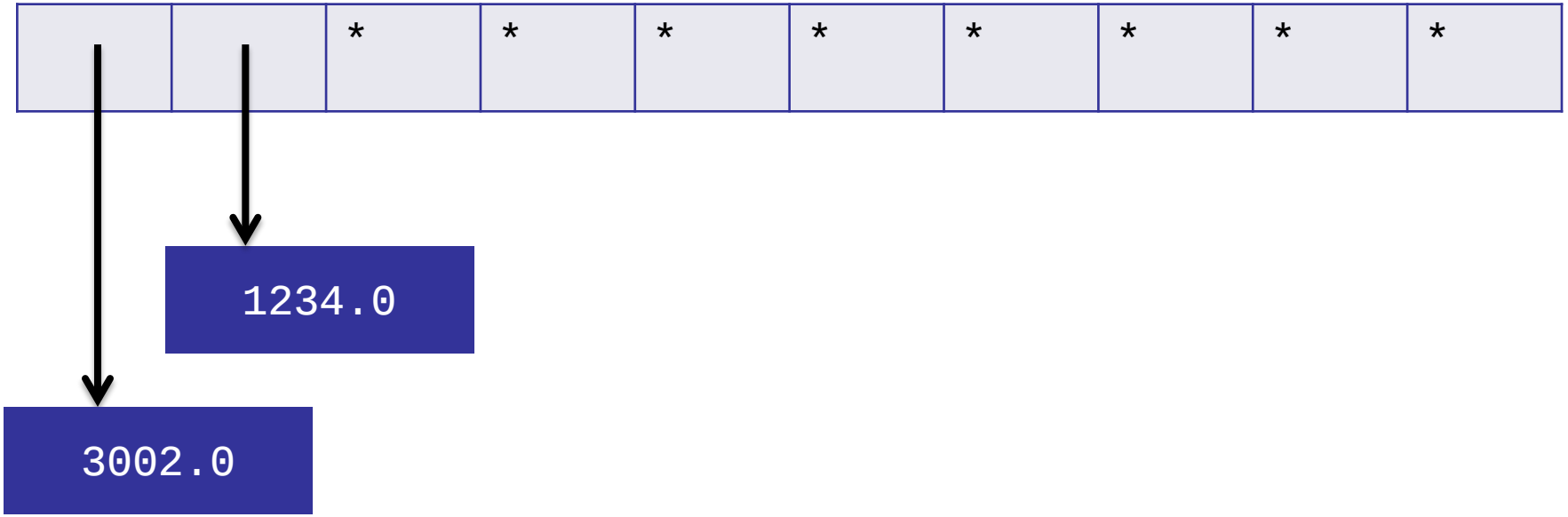
- `double (*q)[10];`



비교

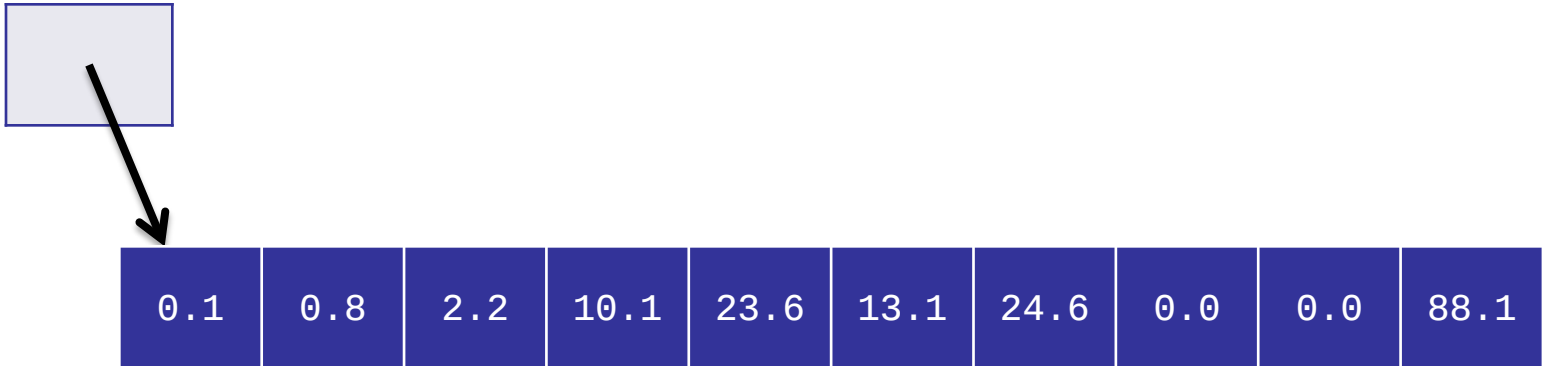
- double x;

- double *p[10];



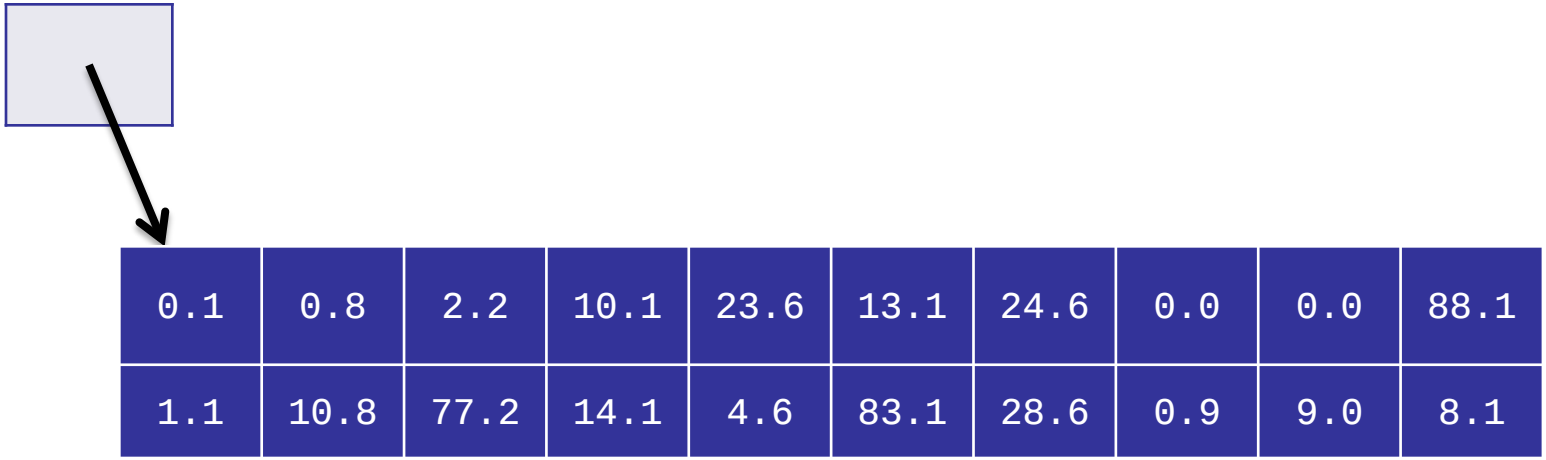
비교

- double (*q)[10];



비교

- `double (*q)[10];`



main() 함수의 인자

- main()의 인자는 운영체제와의 통신을 위해 사용됨
  - 인자의 형과 기능은 고정되어 있음
- int main(int argc, char *argv[])
  - argc : 명령어 라인 인자의 개수를 가짐
  - argv : 명령어 라인을 구성하는 문자열들을 포인트 함

main() 함수의 인자

프로그램 8.11

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    printf("총 인자 개수 : %d\n", argc);
    for (i = 0; i < argc; ++i)
        printf("%d 번째 인자 : %s\n", i, argv[i]);
    return 0;
}
```

프로그램 결과

```
$ echo hello main
총 인자 개수 : 3
0 번째 인자 : echo
1 번째 인자 : hello
2 번째 인자 : main
```

형 한정자

- 변수의 사용 제한 설정
  - const
  - volatile
  - restrict : C99에서 추가
- 기억영역 클래스 뒤와 형 앞에 지정
- const와 restrict

const

- const 변수는 초기화될 수는 있지만, 그 후에 배정되거나, 증가, 감소, 또는 수정될 수 없음
- 예

```
const float pi = 3.14;
                  // pi에는 다른 값을 배정할 수 없음
pi = 3.141592;  // 오류
```

- 배열 예

```
const char months[][10] =
   {"January", "February", "March", "April",
    "May", "June", "July", "August", "September",
    "October", "November", "December"};
```

const

- **const 변수를 포인트 할 때 주의**

```
const int   a = 7;
int        *p = &a;   // 오류
```

  - p는 int를 포인트 하는 보통의 포인터이기 때문에, 나중에 ++*p 와 같은 수식을 사용하여 a에 저장되어 있는 값을 변경할 수 있기 때문

const

- const 변수를 포인트 해야 할 경우

```
const  int   a = 7;
const  int  *p = &a;
```

  - 여기서 p 자체는 상수가 아님
  - p에 다른 주소를 배정할 수 있지만, *p에 값을 배정할 수는 없음

const

- **상수 포인터**

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

  - p는 int에 대한 상수 포인터임
  - p에 값을 배정할 수는 없지만, *p에는 가능함
  - 즉, ++*p, *p = 10 등과 같은 수식은 가능

const

- const 변수에 대한 상수 포인터 선언

```
const int          a = 7;
const int *const   p = &a;
```

  - p는 const int를 포인트하는 상수 포인터임
  - p와 *p 값은 변경이 안됨

restrict

- C99에서 추가
- 포인터 변수에 적용되며, 현재 포인트 되는 객체는 다른 포인터에 의해서는 포인트 안됨을 나타냄
- 컴파일러가 코드 최적화를 할 수 있음
- 예

```
int block_copy(int * restrict dest,
               int * restrict org, int size);
```

  - dest와 org는 완전히 독립된 객체를 포인터함

함수 포인터

- 함수 이름 : 함수 포인터
- 매개변수로 명시된 함수 : 함수 포인터
- 함수 포인터를 사용하면 다양한 함수에 같은 일을 적용할 때 유용함

함수 포인터

- 예

```
void qsort(void *array, size_t n_els,
           size_t el_size,
           int compare(const void *, const void *));
```

  - qsort()는 다양한 형의 배열을 퀵 정렬로 정렬할 수 있게 함
  - el_size 크기의 원소가 n_els개 있는 array 배열을 정렬함
  - 마지막 매개변수인 compare는 함수 포인터임
  - compare는 const void * 형 매개변수를 두 개 갖고 int 형을 리턴 하는 함수를 포인터 할 수 있음

qsort()

- int 형을 위한 compare 함수 예

```
int compare_int(const void *p, const void *q){
    if (*(int *)p > *(int *)q)
        return 1;
    else if (*(int *)p < *(int *)q)
        return -1;
    return 0;
}
```

  - void * 형 포인터인 p와 q를 통해 값을 비교할 때 먼저 (int *)로 캐스트 해야 함

qsort()

```
int compare_int(const void *p, const void *q){
   if (*(int *)p > *(int *)q)
      return 1;
   else if (*(int *)p < *(int *)q)
      return -1;
   return 0;
}
int main(void){
  int i, int_data[10] = {1, 20, 3, 4, 5, 60, 7, 8, 9, 10};
  qsort(int_data, 10, sizeof(int), compare_int);

  for (i = 0; i < 10; i++)
     printf("%d ", int_data[i]);
  printf("\n");

  return 0;
}
```

qsort()

- **qsort() 사용 예**

```
int compare_word(const void *p, const void *q){
   return strcmp(*(char **)p, *(char **)q);
}
. . .
qsort(words_p, num_word, sizeof(char *), compare_word);
```

– compare_word 뒤에 괄호가 없으므로 함수 포인터 임

qsort()

프로그램 8.12 (일부)

```
int compare_word(const void *p, const void *q){
   return strcmp(*(char **)p, *(char **)q);
}

int main(void){
   int i;
   char *words_p[N] = {NULL};
   int i, num_word = 0;
   num_word = input_words(words_p);
   qsort(words_p, num_word, sizeof(char *), compare_word);
   print_words(words_p, num_word);
   . . .
   return 0;
}
```

함수 포인터 예제

```
#include <stdio.h>
void func_1(void){
   printf("I am func_1.\n");
}
void func_2(void){
   printf("I am func_2.\n");
}

int main(void){
   void (* func)(void);
   func = func_1;
   func();

   func = func_2;
   func();
   return 0;
}
```

함수 포인터 예제

```
I am func_1.
I am func_2.
```