



# 제1장

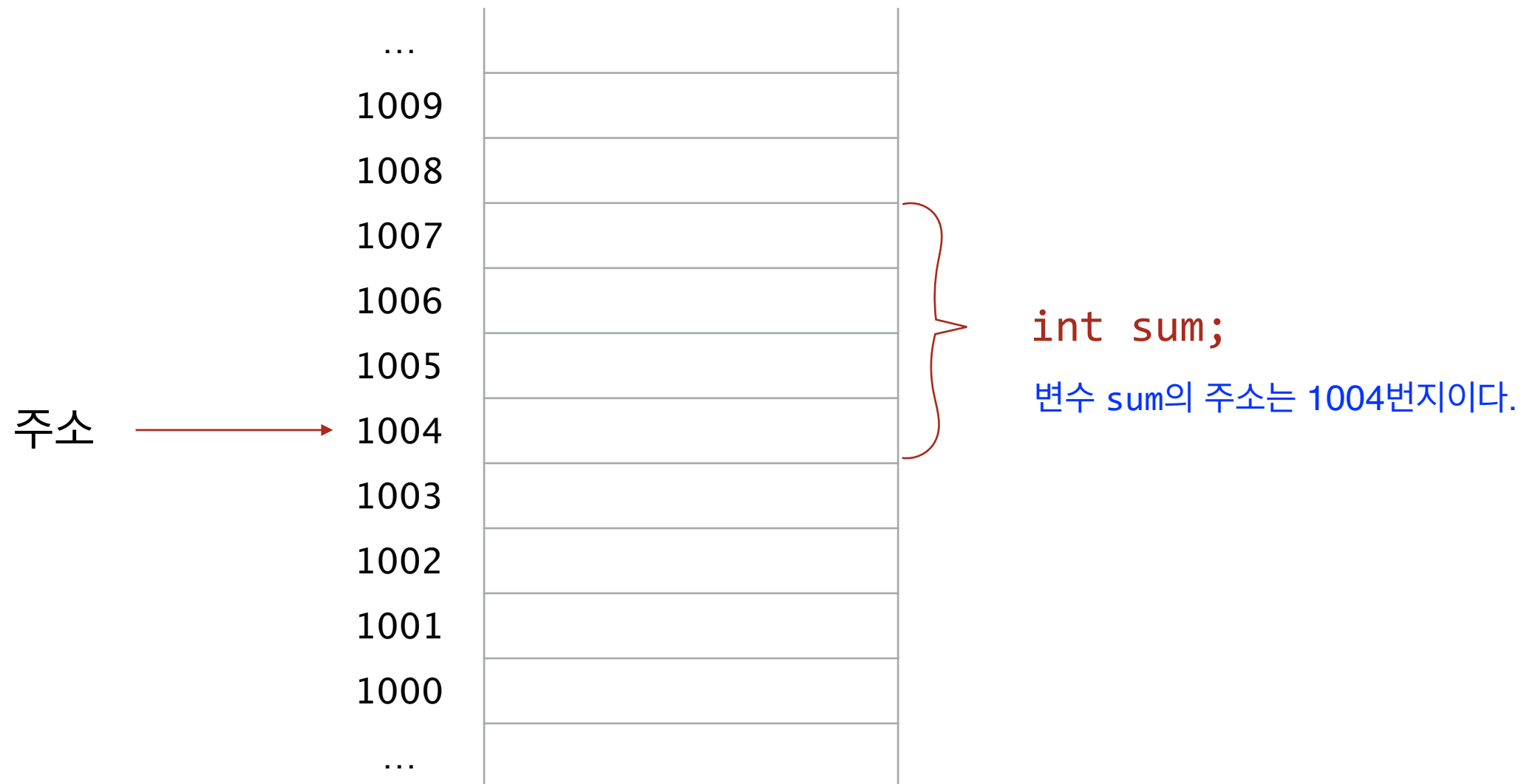
## C 언어 리뷰

# 기초 문법 리뷰

배열, 포인터, 문자열, 동적메모리할당

# 메모리

- 컴퓨터의 메모리는 데이터를 보관하는 장소
- 바이트(8 bits) 단위로 주소가 지정됨
- 모든 변수는 주소를 가짐



- 포인터(pointer)는 메모리 주소를 값으로 가지는 변수이다. 포인터 변수는 다음과 같이 선언된다.

`type-name * variable-name;`

- `variable-name`은 선언된 포인터 변수의 이름이며, `*`는 `variable-name`이 포인터 변수임을 표시하고, `type-name`은 포인터 변수 `variable-name`에 저장될 주소에 저장될 데이터의 유형을 지정

`int * ptr;`

정수형                  포인터변수

# 포인터

- 연산자 &는 변수로부터 그 변수의 주소를 추출하는 연산자이다.

```
int c = 12;  
int *p;  
p = &c;
```

포인터변수 p에 변수 c의  
주소를 저장한다.

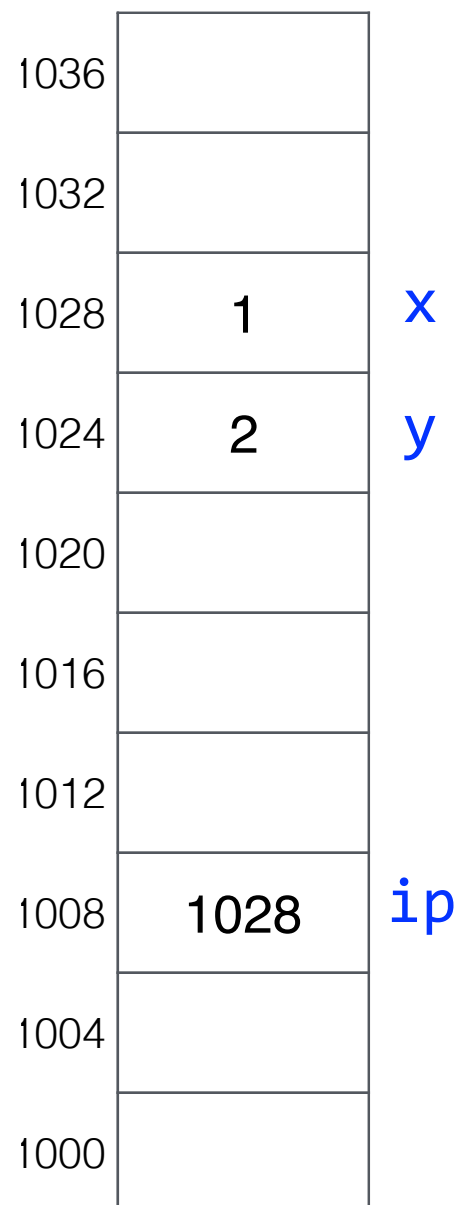
1036		
1032		
1028	1008	p
1024		
1020		
1016		
1012		
1008	12	c
1004		
1000		

# 포인터

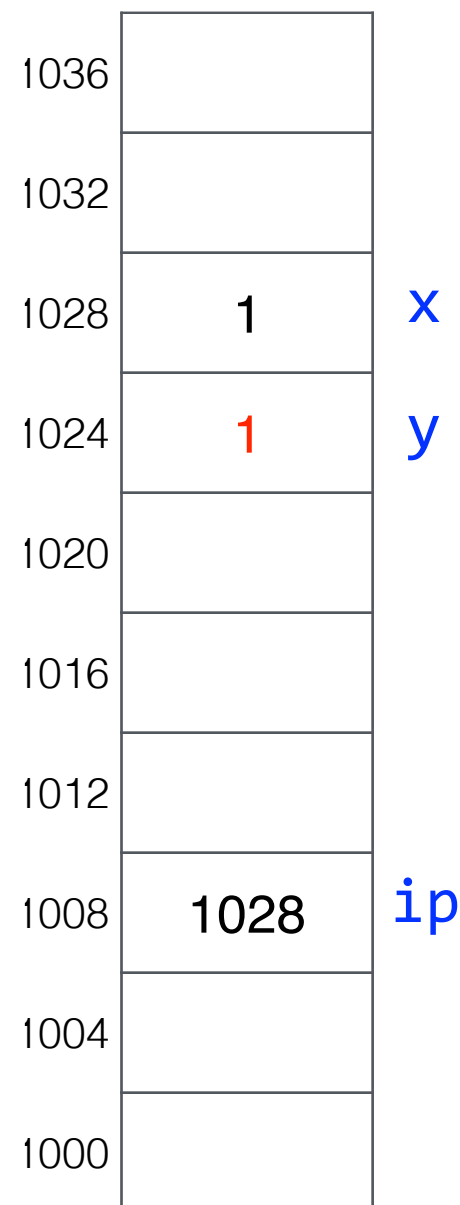
```
int x=1, y=2;
int *ip;
ip = &x;

y = *ip;

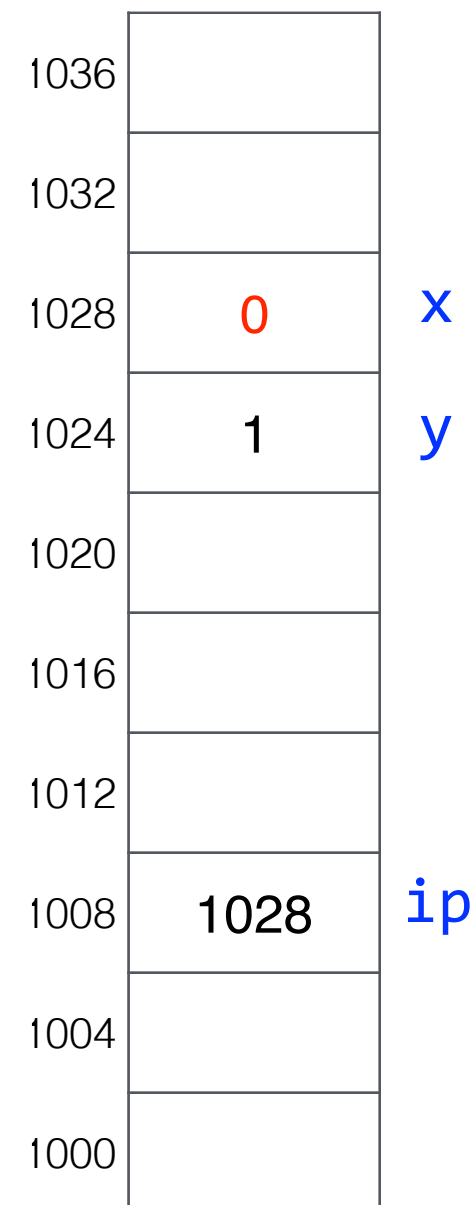
*ip = 0;
```



```
int x=1, y=2;
int *ip;
ip = &x;
```



```
y = *ip;
```



```
*ip = 0;
```

# 포인터와 배열

- 포인터와 배열은 매우 긴밀히 연관되어 있다.
- 예를 들어 다음과 같이 선언된 배열 `a`가 있다고 하자.

```
int a[10];
```

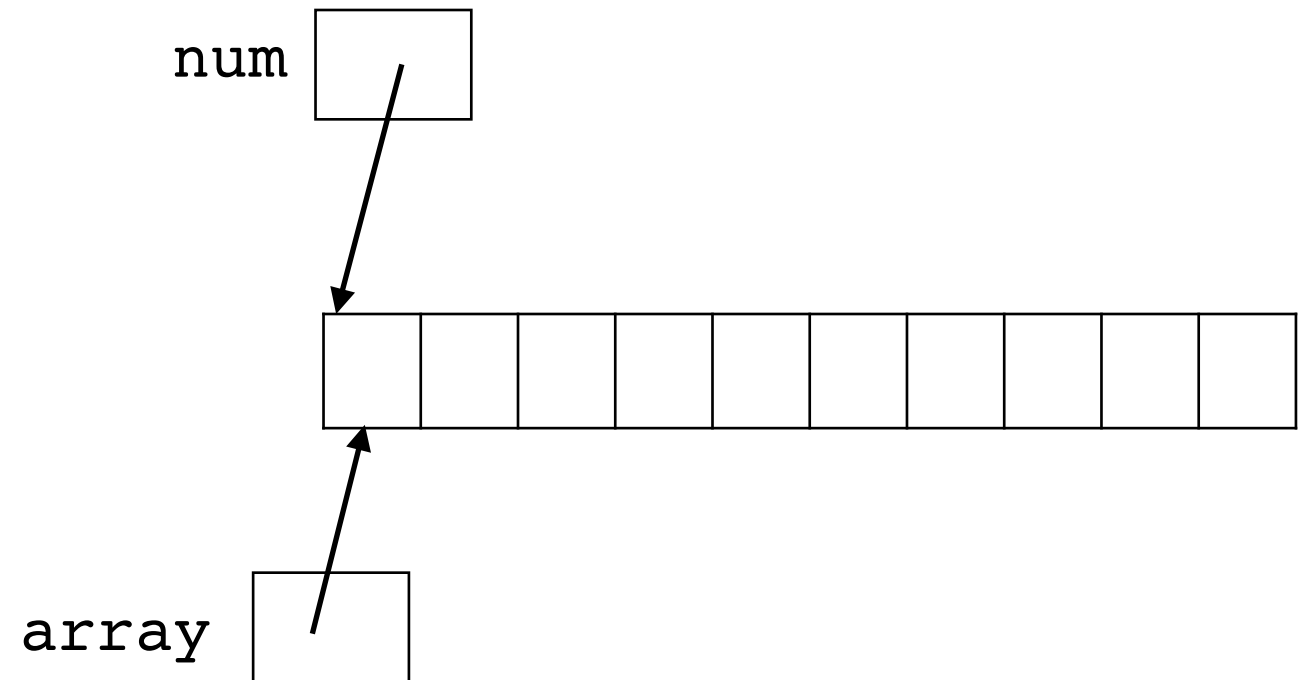
배열의 이름은  
배열의 시작 주소를 저장하  
는 포인터 변수임  
(단 그 값을 변경할 수 없음)



# 예제

```
#include <stdio.h>
int main(void)
{
    int sum, average;
    int num[10];
    for ( int i = 0; i < 10; i++ )
        scanf("%d", &num[i]);
    sum = calculate_sum( num );
    average = sum / 10;
    printf("%d\n", average);
    return 0;
}

int calculate_sum( int *array )
{
    int sum = 0;
    for ( int i = 0; i < 10; i++ )
        sum = sum + array[i];
    return sum;
}
```



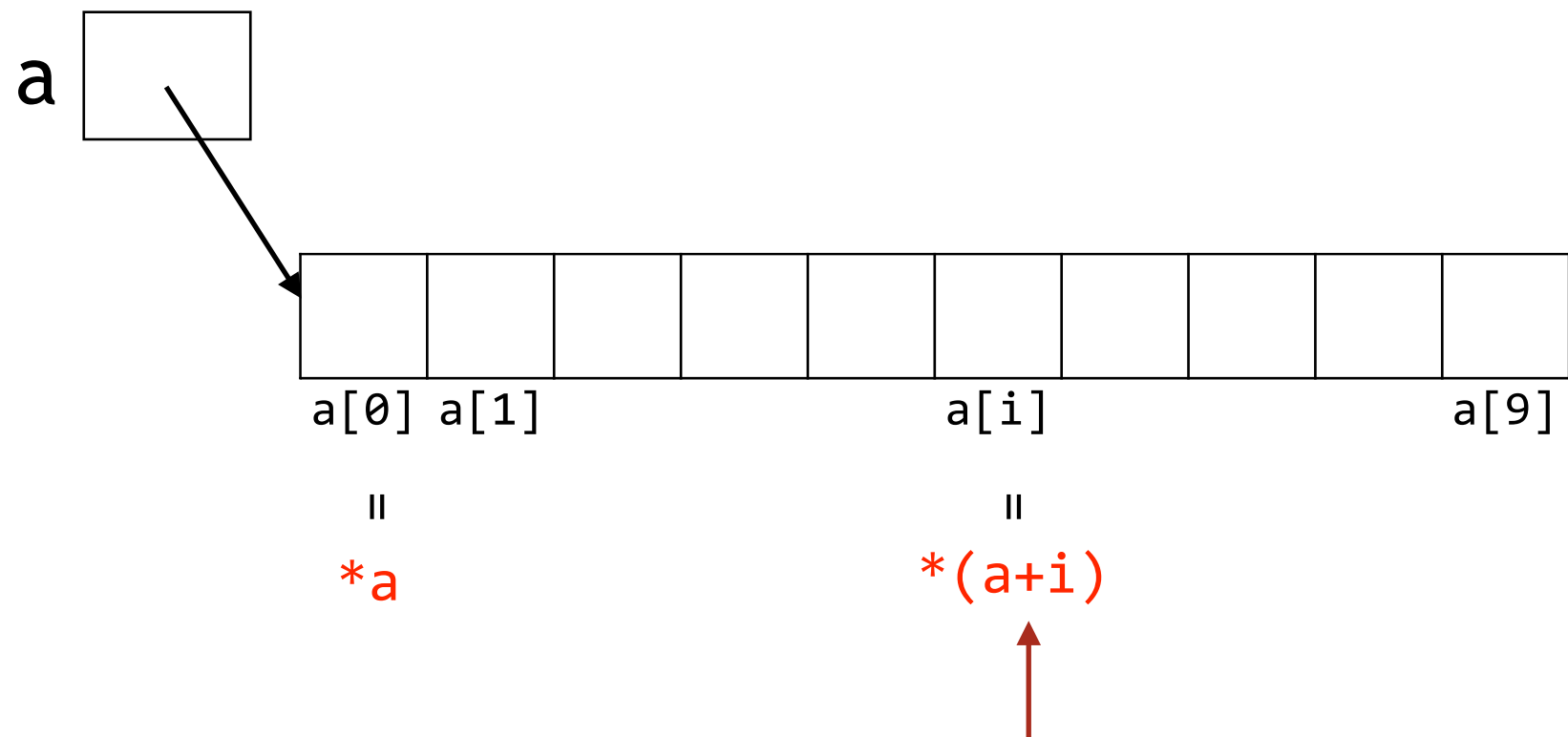
배열을 매개변수로 받을 때  
`int array[]`  
대신 이렇게 포인터로 받을 수도 있다.



# 포인터 arithmetic

- $*a$ 와  $a[0]$ 은 동일한 의미이다.
- 또한  $a[1]$ 은  $*(a+1)$ 과 동일하고,  $a[i]$ 는  $*(a+i)$ 와 동일하다.

```
int a[10];
```



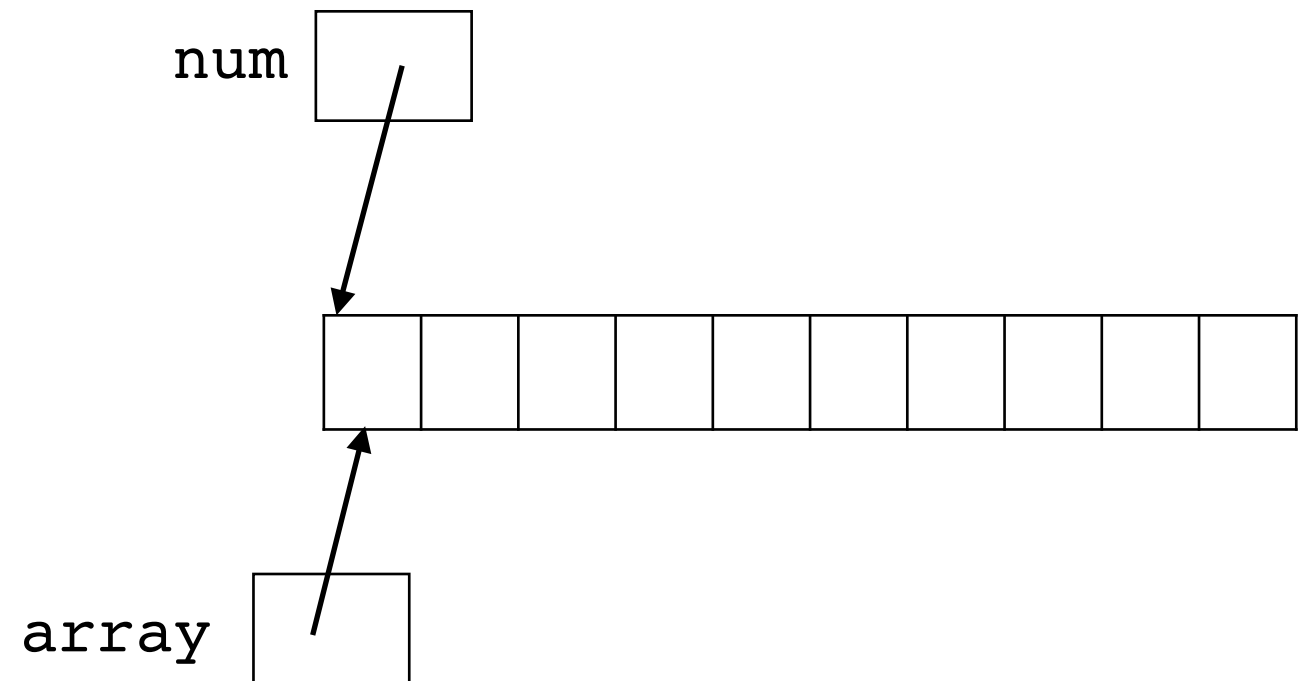
이런 연산을 포인터 arithmetic이라고 부른다.

# 포인터 arithmetic

```
#include <stdio.h>
int main(void)
{
    int sum, average;
    int num[10];
    for ( int i = 0; i < 10; i++ )
        scanf("%d", &num[i]);
    sum = calculate_sum( num );
    average = sum / 10;
    printf("%d\n", average);
    return 0;
}

int calculate_sum(int * array)
{

```



## 동적메모리 할당

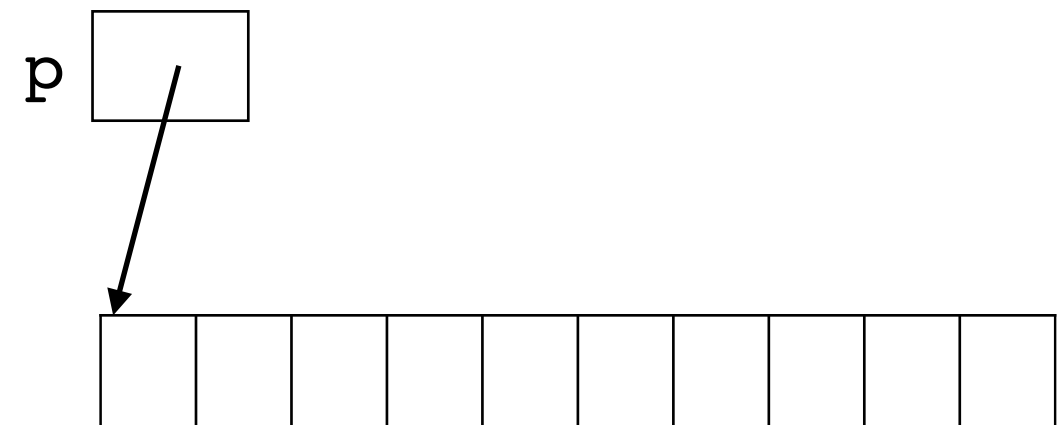
- 변수를 선언하는 대신 프로그램의 요청으로 메모리를 할당할 수 있다. 이것을 **동적 메모리 할당**(dynamic memory allocation)이라고 부른다.
- malloc 함수를 호출하여 동적메모리할당을 요청하면 요구하는 크기의 메모리를 할당하고 그 **시작 주소를 반환**한다.

# malloc 함수

malloc이 반환하는 주소는 타입이 없는 주소(void \*)이다. 정수들을 저장하기 위해서 이것을 int \*로 변환한다. 반드시 필요한 건 아니다.

할당받을 메모리의 크기를 byte단위로 지정한다. 여기서는 10개의 정수를 저장하기 위해서 40바이트를 요청하였다.

```
int * p;  
p = (int *)malloc(40);  
if (p==NULL) {  
    /* 동적 메모리 할당이 실패 */  
    /* 적절한 조치를 취한다. */  
}
```



```
p[0] = 12;  
p[1] = 24;  
*(p+2) = 36;  
...
```

← malloc으로 할당받은 메모리는 이렇게 보통의 배열처럼 사용한다.

# 배열 키우기

- 동적으로 할당된 배열은 공간이 부족할 경우 더 큰 배열을 할당하여 사용할 수 있다.

```
int * array = (int *)malloc(4*sizeof(int));
```

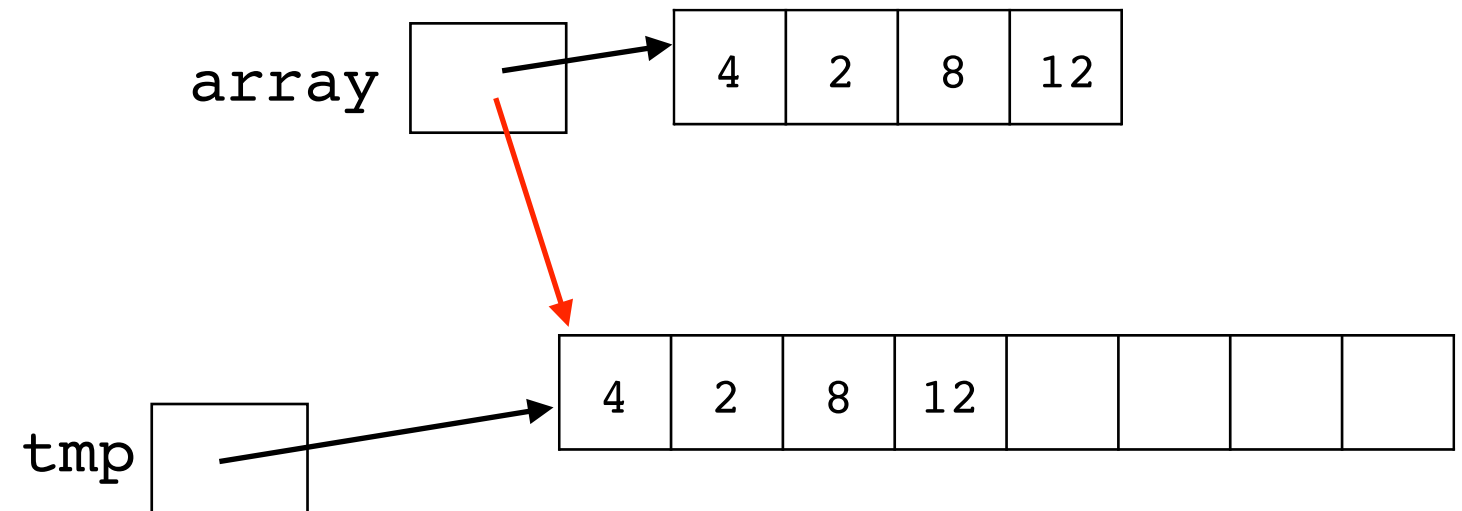
```
/* 배열 array의 크기가 부족한 상황이 발생한다. */
```

```
int * tmp = (int *)malloc(8*sizeof(int));
```

```
int i;
```

```
for (i=0; i<4; i++)  
    tmp[i] = array[i];
```

```
array = tmp;
```



# 문자열 (string)

- 문자열은 char타입의 배열의 각 칸마다 문자 하나씩 저장됨

```
char str[6];  
str[0] = 'h';  
str[1] = 'e';  
str[2] = 'l';  
str[3] = 'l';  
str[4] = 'o';  
str[5] = '\0';
```

null character(‘\0’)는 문자열의 끝을 표시하는 역할을 한다.  
즉 배열의 크기가 문자열의 길이보다 적어도 1만큼 길어야 한다.

str

h	e	l	l	o	\0
---	---	---	---	---	----

- C 언어는 문자열을 생성하는 편리한 방법을 제공

```
char str[] = "hello";
```

혹은

```
char *str = "hello";
```

하지만 이렇게 정의된 문자열은 수정이 불가능하다는 점에서 위의 두 방법과 다르다. 이것을 string literal이라고 부른다.

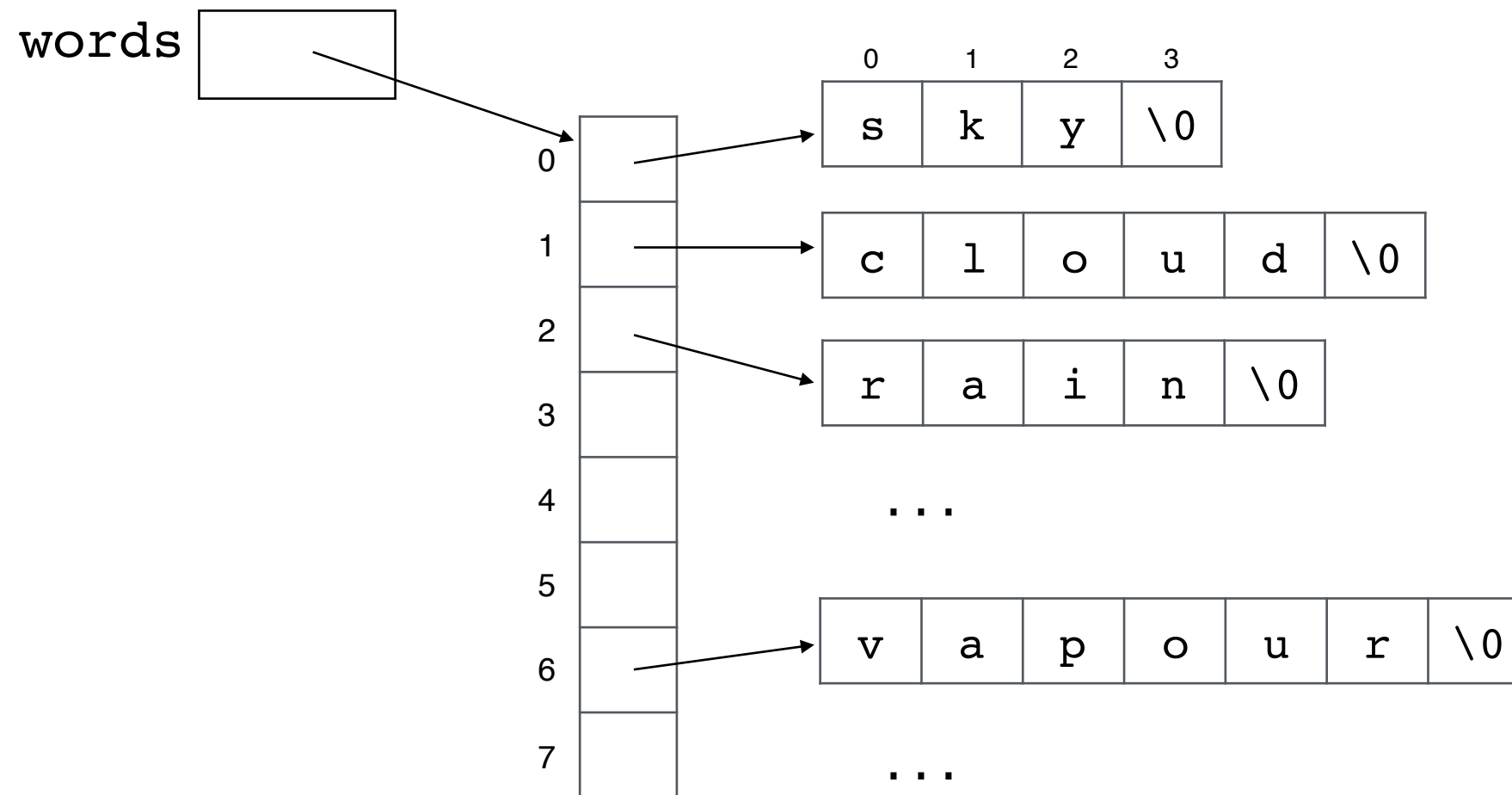
# string.h 라이브러리 함수

- string.h 라이브러리는 문자열을 다루는 다양한 함수를 제공

strcpy	문자열 복사
strlen	문자열의 길이
strcat	문자열 합치기
strcmp	문자열 비교

## 문자열들의 저장

- 여러개의 단어들을 포인터를 이용하여 아래 그림과 같이 저장해보자.





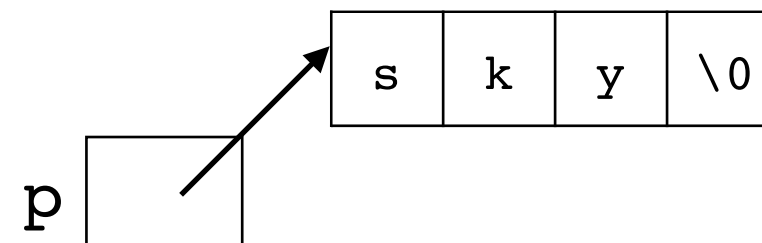
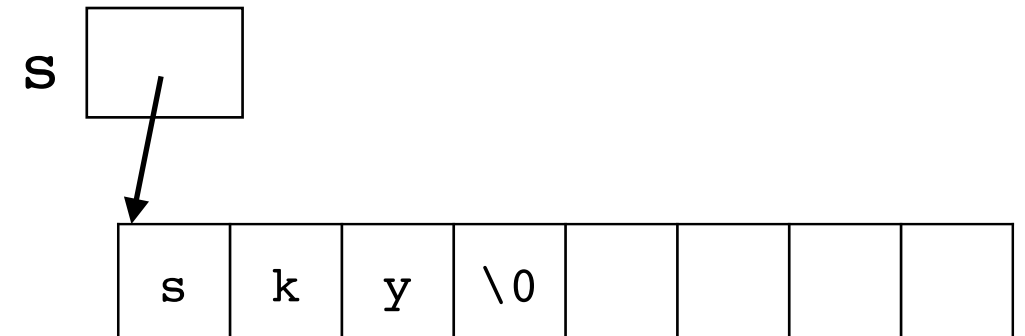
## 단어들 입력받아 저장하기

```
char * words[MAXWORDS];  
int nwords;  
  
char tmp[MAXLEN];  
nwords = 0;  
  
while (scanf("%s", tmp) != EOF) {  
  
}
```

# 문자열 복사: strdup

매개변수로 받은 하나의 문자열을 복제하여 반환한다.

```
char * strdup(char *s)
{
}
}
```



strcpy와의 차이는 ?

# 파일로부터 읽기

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

```
void main() {
```

```
}
```

# 파일 읽고 쓰기

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

```
void main() {
```

```
}
```

## 실습 문제

# 연습 1

프로그램을 실행하면 화면에 프롬프트(\$)와 한 칸의 공백문자를 출력하고 사용자의 입력을 기다린다.

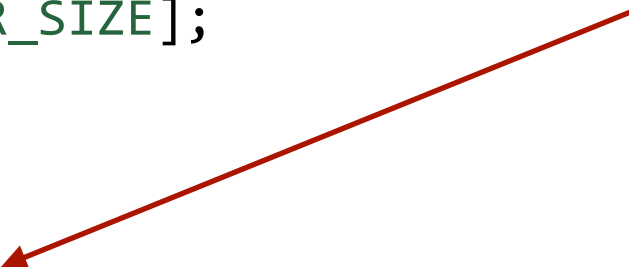
↓  
\$ hello↵ ← 문장을 입력하고 리턴(↵) 키를 친다.  
hello:5 ← 리턴(↵) 을 제외하고 입력한 문장을 그대로 출력하고 입력한 문장의 길이를 출력한다.  
\$ welcome to the class↵  
welcome to the class:20 ← 공백문자도 포함하여 카운트한다.  
\$       programming is fun, right?       ↵  
      programming is fun, right?       :35 ← 문장의 앞뒤에 붙은 공백까지 그대로 출력해야 한다.

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

#define BUFFER_SIZE 100

int main() {
    char buffer[BUFFER_SIZE];
    while (1) {
        printf("$ ");
        scanf("%s", buffer);
        printf("%s:%d\n", buffer, strlen(buffer));
    }
    return 0;
}
```

프롬프트를 출력한다.



# gets

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

```
#define BUFFER_SIZE 10
```

← 배열의 크기보다 더 긴 문장을 입력해본다.  
어떤 문제가 생기는지 확인한다.

```
int main() {
```

```
    char buffer[BUFFER_SIZE];
```

```
    while (1) {
```

```
        printf("$ ");
```

```
        gets(buffer); ← gets 함수는 라인을 통채로 읽는다.
```

```
        printf("%s:%d\n", buffer, strlen(buffer));
```

```
    }
```

```
    return 0;
```

```
}
```



# fgets

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

```
#define BUFFER_SIZE 100
```

```
int main() {
```

```
    char buffer[BUFFER_SIZE];
```

```
    while (1) {
```

```
        printf("$ ");
```

```
        fgets(buffer, BUFFER_SIZE, stdin);
```

```
        printf("%s:%d\n", buffer, strlen(buffer));
```

```
    }
```

```
    return 0;
```

```
}
```

stdin은 표준 입력, 즉 키보드를 의미한다.



문제가 생기는 이유와 해결방법은 ?

```
#include <stdio.h>
#include <string.h>
#define BUFFER_SIZE 100

int main() {
    char buffer[BUFFER_SIZE];
    int k;
    while (1) {
        printf("$ ");
        k=read_line(buffer, BUFFER_SIZE);
        printf("%s:%d\n", buffer, k);
    }
    return 0;
}

int read_line( char str[], int n )    {

}
```

## 연습 2

공백문자들이 문장의 앞, 중간, 뒤에 포함되어 있다.

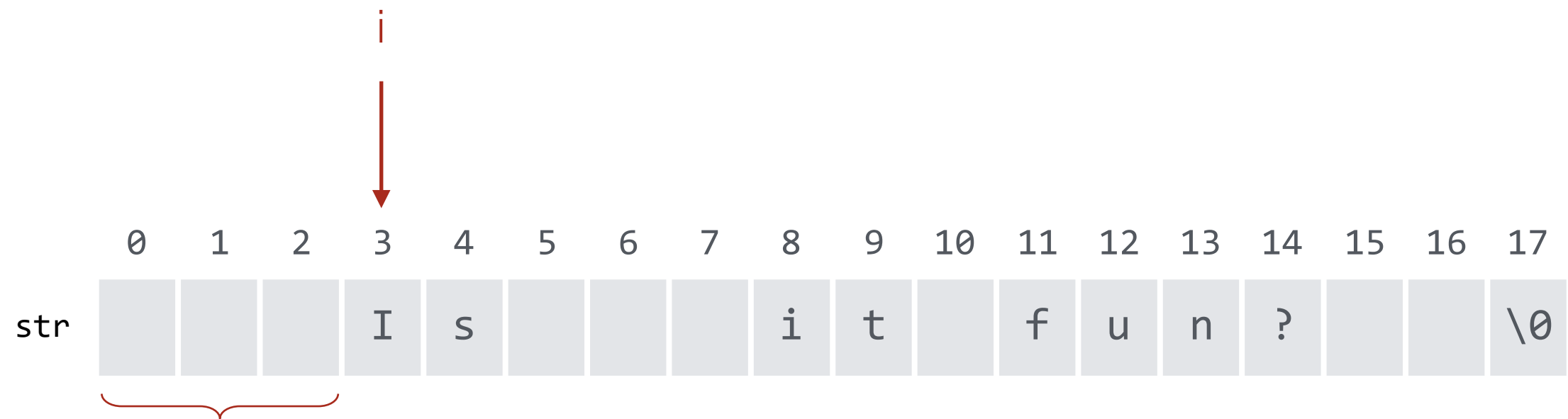
```
$    hello ↵  
hello:5  
$ welcome  to the class ↵  
welcome to the class:20 ↵  
$  programming is    fun, right?  ↵  
programming is fun, right?:26 ↵
```

문장의 앞과 뒤에 붙은 공백문자들은 제거하고  
단어 사이에 두 개 이상의 연속된 공백문자들은 하나의 공백 문자로 대체하라.

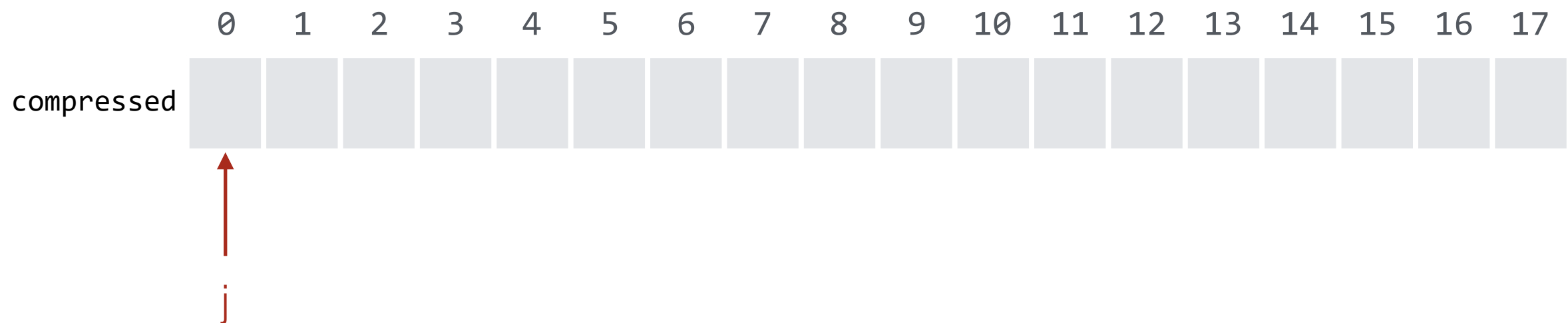
# 압축하기



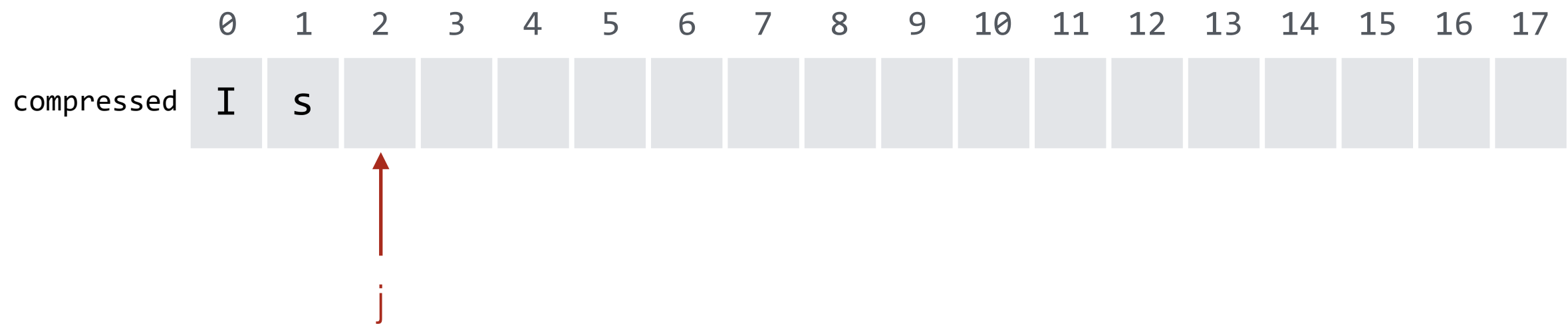
# 압축하기



맨 앞의 공백들을 건너뛴다.



# 압축하기



# 압축하기



## 압축하기





'\0'-terminated string



**compress**

```
int compress_with_additional_array( char str[] ) {
```

```
}
```

# compress

'\0'-terminated string



```
int compress( char str[] ) {
```

```
}
```

## compress while reading

```
int read_line_with_compression( char str[], int n ) {
```

```
}
```