C언어 기초 part. 2

Week 3 – 포인터와 메모리

QnA 메일 : edujongkook@gmail.com

Pdf 파일 : github.com/edujongkook
/pdf_sbs_c_weekend

목차 A table of contents

1 배열과 포인터 연산

2 배열 포인터

3 포인터 배열

4 동적 메모리 할당

1.

배열과 포인터

1 배열의 이름

배열의 이름에는 크기뿐 아니라 첫번째 원소의 주소값도 가지고 있습니다.

```
int main(void) {
    int a[2] = { 10, 20 };
    printf("%d₩n", sizeof(a));
    printf("%p₩n", a);
    printf("%p₩n", &a[0]);
}
```

1 배열의 이름

따라서 &없이 바로 포인터에 대입 할 수 있고

연산자 * 를 사용하여 값에 접근할 수 있습니다.

```
int main(void) {
    int a[2] = { 10, 20 };
    int* p_a = a;
    printf("%d₩n",*a);
}
```

1 배열과 포인터 차이

배열은 상수처럼 새로운 값을 대입할 수 없고

포인터는 배열의 크기값을 알 수 없다는 차이점도 있습니다.

```c
int main(void) {
    int a[2] = { 10, 20 };
    int* p_a = a;
    int b[2] = { 3, 4 };
    a = b; // 배열은 새로운 대입이 불가능
    p_a = b; // 포인터는 변수처럼 사용
    printf("%d₩n", sizeof(p_a)); // 포인터는 크기값 고정
}
```

1 배열과 포인터 차이

배열은 상수처럼 새로운 값을 대입할 수 없고

포인터는 배열의 크기값을 알 수 없다는 차이점도 있습니다.

```c
int main(void) {
    int a[2] = { 10, 20 };
    int* p_a = a;
    int b[2] = { 3, 4 };
    a = b; // 배열은 새로운 대입이 불가능
    p_a = b; // 포인터는 변수처럼 사용
    printf("%d₩n", sizeof(p_a)); // 포인터는 크기값 고정
}
```

1 배열과 포인터 인덱스

배열과 포인터 모두 인덱스를 통해 값에 접근할 수 있고

```
int main(void) {
    int a[2] = { 10, 20 };
    int* p_a = a;
    printf("%d₩n", a[1]);
    printf("%d₩n", p_a[1]);
}
```

1 포인터 연산

배열과 포인트 모두 포인터 연산이란것이 가능합니다.

```
int main(void) {
    int a[2] = { 10, 20 };
    int* p_a = a;
    printf("%d₩n", *(a + 1));
    printf("%d₩n", *(p_a + 1));
}
```

1 포인터 연산

배열과 포인트 모두 포인터 연산이란것이 가능합니다.

```
int main(void) {
    int a[2] = { 10, 20 };
    int* p_a = a;
    printf("%d₩n", *(a + 1));
    printf("%d₩n", *(p_a + 1));
}
```

1 포인터 연산

포인트 연산이란 주소값에 더하거나 빼는 연산을 통해 주소값을 계산합니다.

(곱셈, 나눗셈 제외) 이때 숫자 1의 크기는 자료형의 크기입니다.

```
int main(void) {
    int a[3] = { 10, 20 };
    int* p_a = a;
    printf("%d₩n", p_a);
    printf("%d₩n", p_a + 1);
    printf("%d₩n", p_a - 1);
}
```

1 포인터 연산

주소값들 사이에 차이도 계산할 수 있습니다.

이때 결과는 두 주소 사이에 몇 개의 자료형이 들어갈 수 있는지 입니다.

```c
int main(void) {
    int a[3] = { 10, 20 };
    int* p_a = a;
    printf("%d₩n", &p_a[0]);
    printf("%d₩n", &p_a[1]);
    printf("%d₩n", &p_a[1] - &p_a[0]);
    printf("%d₩n", &p_a[0] + &p_a[1]); // 포인터간 합은 불가능
}
```

1 포인터 연산

그 밖에도 주소값이 서로 같은지 비교하거나 어떤 주소값이
더 큰지 작은지 비교하는 연산이 가능합니다.

```
int main(void) {
    int a[2] = { 10, 20 };
    int* p_a = a;
    printf("%d₩n", p_a == a);  // 결과 1
    printf("%d₩n", p_a > p_a + 1); // 0
    printf("%d₩n", p_a - 1 < p_a); // 1
}
```

1 포인터 연산

포인터 연산은 인덱스를 사용하는 방법과 비슷합니다.

(인덱스 연산은 내부적으로 포인터연산으로 처리됩니다.)

```
int main(void) {
    int a[2] = { 10, 20 };
    int* p_a = a;
    printf("%d₩n", p_a[1]);
    printf("%d₩n", *(p_a + 1));
    printf("%d₩n", p_a[2]); // 배열범위 벗어남
    printf("%d₩n", *(p_a + 2));
}
```

1 포인터 연산

다만 배열은 대입연산, 증감연산이 불가능합니다.

```
int main(void) {
    int a[2] = { 10, 20 };
    int* p_a = a;
    printf("%d₩n", *(p_a++));
    printf("%d₩n", *(a++)); // 배열은 대입이 불가능
}
```

2 매개변수로의 배열

배열을 매개변수로 받는 함수는 사실 배열 전체를 복사하는 것이 아니고
그 주소값만 복사해서 전달합니다.

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

2 매개변수로의 포인터

따라서 아래와 같이 포인터 변수로 매개변수를 선언해도

동일한 의미 입니다. 아래의 코드를 포인터 연산으로 바꿔보세요

```
int sum_array(int* nums, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += nums[i];
    }
    return sum;
}
```

2.

배열 포인터

2 매개변수로의 포인터

포인터 연산을 사용하면 수식을 단순하게 표현할 수 있게 되지만
익숙해지기 전까지는 조금 혼동이 올 수 있습니다.

```
int sum_array(int* nums, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += *(nums + i);
    }
    return sum;
}
```

2 다차원 배열 매개변수

다음과 같은 2차원 배열을 함수의 매개변수로 전달하여

총합을 구할 수 있게 sum_array 함수 선언을 수정해보겠습니다.

```
int sum_array(int nums[][3], int size);

int main(void) {
    int a[2][3] = { {10, 20, 30}, {40, 50, 60} };
    printf("sum = %d₩n",sum_array(a, 2));
}
```

다차원 배열 매개변수

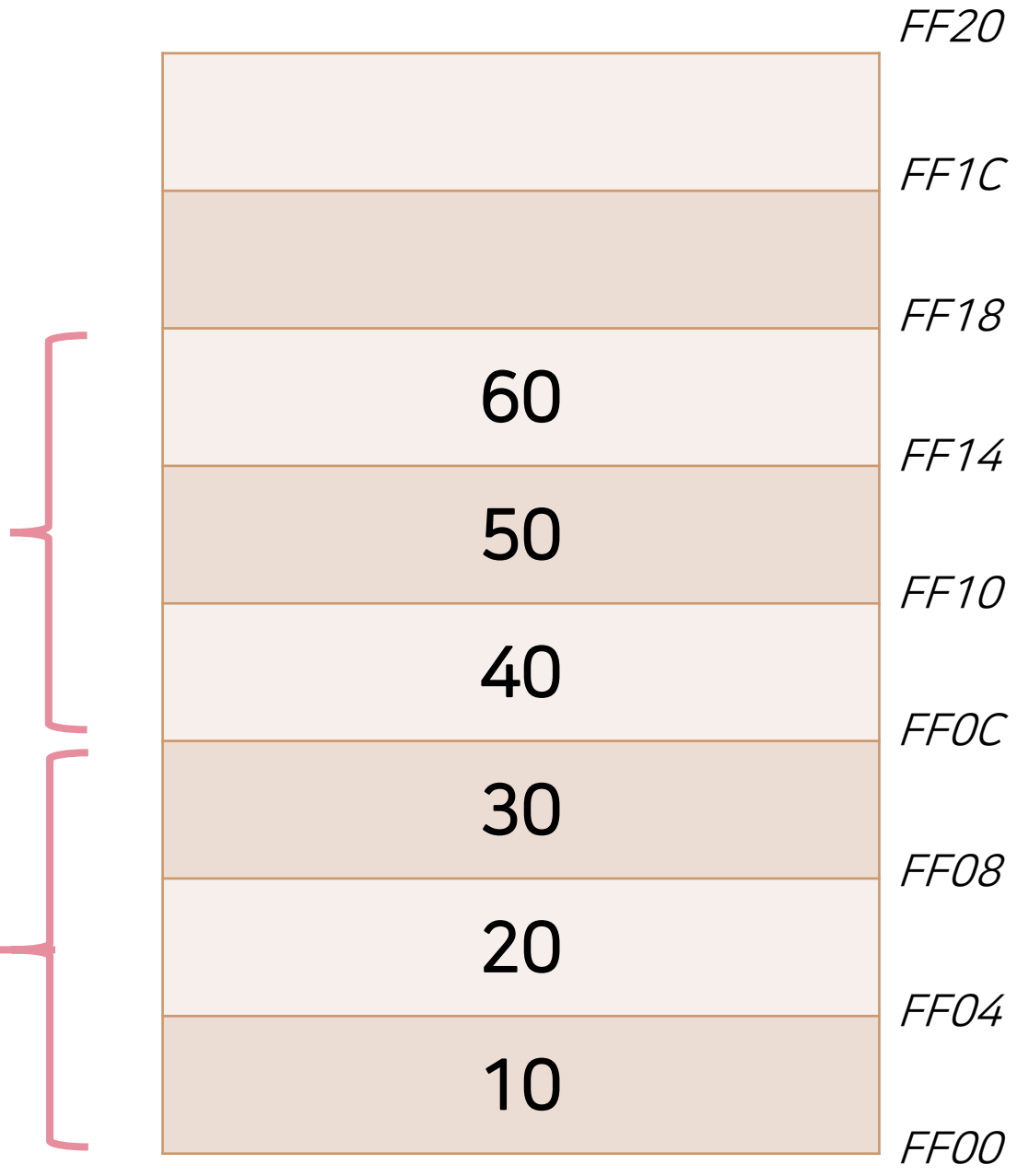
sum_array 매개변수 부분이 변하는데 최상위 차원의 개수는 없어도 되지만
나머지 차원의 개수정보는 주어져야 합니다.

```
int sum_array(int nums[][3], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < 3; j++) {
            sum += nums[i][j];
        }
    }
    return sum;
}
```

2 배열 포인터

이 때 함수로 전달되는 포인터는
첫번째 원소의 주소를 가지는 포인터 입니다.
(첫번째 원소 : 3개의 값을 가지는 배열)
이를 배열 포인터 라고 부릅니다.

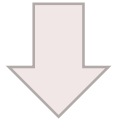
```
int sum_array(int nums[][3], int size)
```



2 배열 포인터

배열 포인터는 다음과 같이 선언할 수도 있습니다.

```
int sum_array(int nums[][3], int size)
```

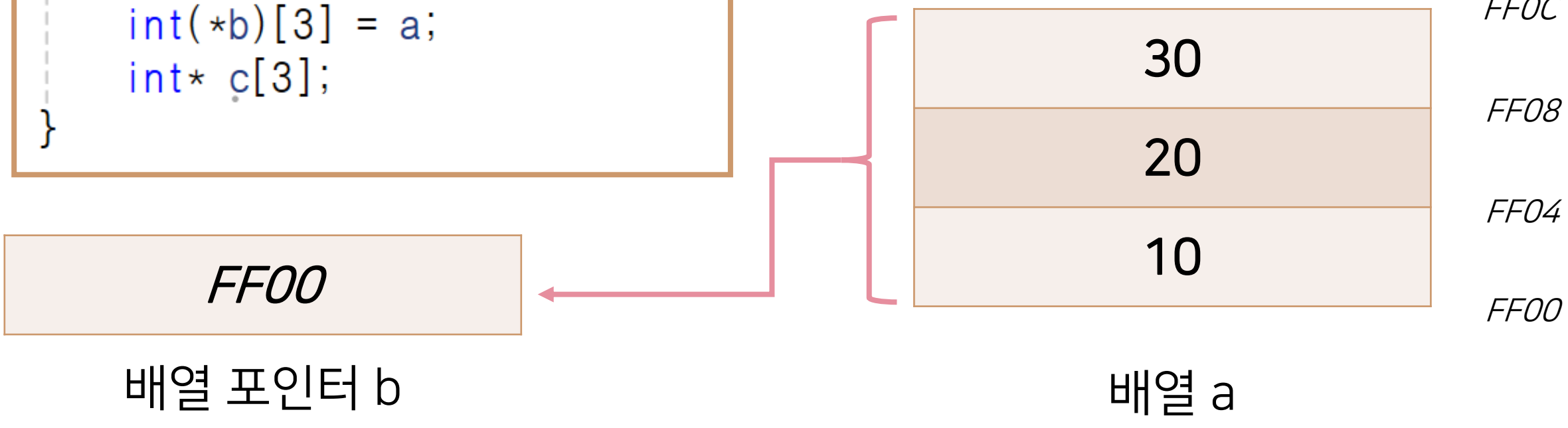


```
int sum_array(int (*nums)[3], int size);
```

배열 포인터

아래의 선언은 모두 서로 다른 의미를 가지게 됩니다.

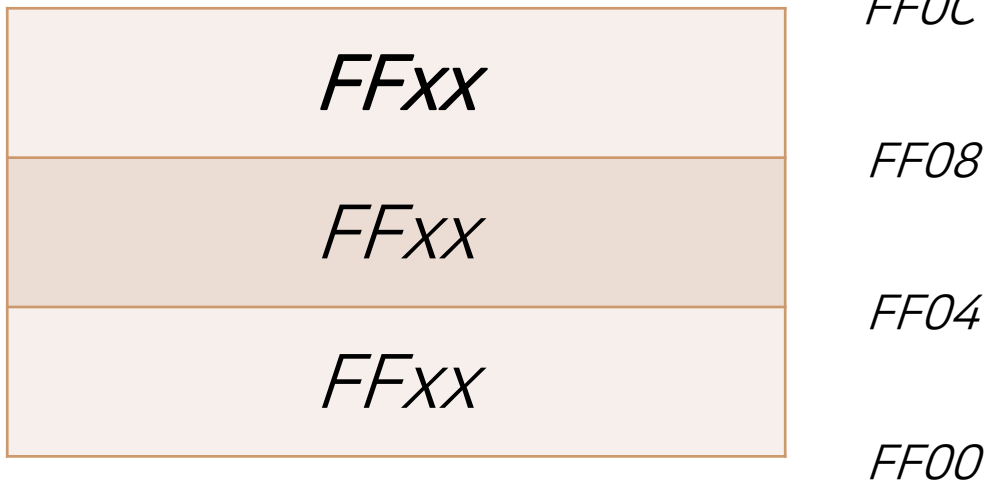
```
int main(void){
    int a[3] = { 10, 20, 30 };
    int(*b)[3] = a;
    int* c[3];
}
```



배열 포인터

int* c [3] 는 주소값을 3개 가지는 배열입니다. 이를 포인터 배열이라고 합니다.

```
int main(void){
    int a[3] = { 10, 20, 30 };
    int(*b)[3] = a;
    int* c[3];
}
```



포인터 배열 c

3.

포인터 배열


IVE THE LIFE YOU'VE
LWAYS DREAMED OF.
BE FEARLESS IN THE
FACE OF ADVERSITY.
EVER STOP LEARNING.
SE YOUR IMAGINATION
WHENEVER POSSIBLE.
ECOGNIZE THE BE AUTY
THAT SURROUN
REMEMBER WHI
AME FROM, BU
LOSE SIGHT OF
YOU ARE GO

3 문자열과 포인터

문자열도 마찬가지로 포인터와 배열이 비슷하게 사용됩니다.

```
int main(void) {
    char str[4] = "abc";
    char* p_s = str;
    printf("%s₩n", str);
    printf("%s₩n", p_s);
}
```

3 문자열과 포인터

문자열 포인터는 특이하게 이런식의 초기화가 가능합니다.

"def" 라는 문자열을 저장할 수 있는 특수한 메모리 영역이 존재 하기 때문입니다.

```
int main(void) {
    char str[4] = "abc";
    char* p_s = "def";
    printf("%s₩n", str);
    printf("%s₩n", p_s);
}
```

문자열과 포인터

하지만 특수한 메모리영역(TEXT영역) 에 저장된 값은 수정할 수 없습니다.

```
int main(void) {
    char str[4] = "abc";
    char* p_s = "def";
    char* p = str;
    p[0] = 'm';
    p_s[0] = 'h'; // 수정 불가
}
```

3 매개변수로의 문자열 포인터

문자열의 개수를 구하는 함수 strsize(char* s) 를 만들어 보겠습니다.

문자열은 숫자배열과 다르게 마지막에 '₩0' 널문자가 있어서 끝을 알 수 있습니다.

```
int strsize(char* s);
int main(void) {
    char* str = "apple";
    printf("%d", strsize(str));
}
int strsize(char* s) {

}
```

3 매개변수로의 문자열 포인터

아래는 자체 제작한 strsize 함수의 예시입니다.

```
int strsize(char* s) {
    int n = 0;
    while (1) {
        if (s[n] == '₩0') {
            return n;
        }
        n += 1;
    }
}
```

3 매개변수로의 문자열 포인터

아래는 표준 라이브러리 string.h 에 선언 되어있는 strlen 함수 코드입니다.

register라는 키워드 외에는 지금까지 다룬 내용입니다. 한번 분석해 봅시다.

```
unsigned int strlen(const char* str) {
    register const char* s;
    for (s = str; *s; s++);
    return (s - str);
}
```

3 문자열의 2차배열

여러 개의 이름이 들어있는

문자열로 이루어진 2차배열을 생각해 봅시다.

```
int main(void) {

    char names[3][6] = { "PARK", "MESSI", "SON" };

}
```

문자열의 2차배열

그림으로 표현하면 아래와 같은 직사각형의 2차원 배열로

각 줄에 문자로 이루어진 6칸의 배열이 있습니다.

P	A	R	K	₩0	
M	E	S	S	I	₩0
S	O	N	₩0		

문자열의 2차배열

이렇게 배열로만 데이터를 쌓다 보면 낭비되는 공간과 크기 제한이 생깁니다.

P	A	R	K	₩0	
M	E	S	S	I	₩0
S	O	N	₩0		

배열만 사용하면

낭비되는 공간

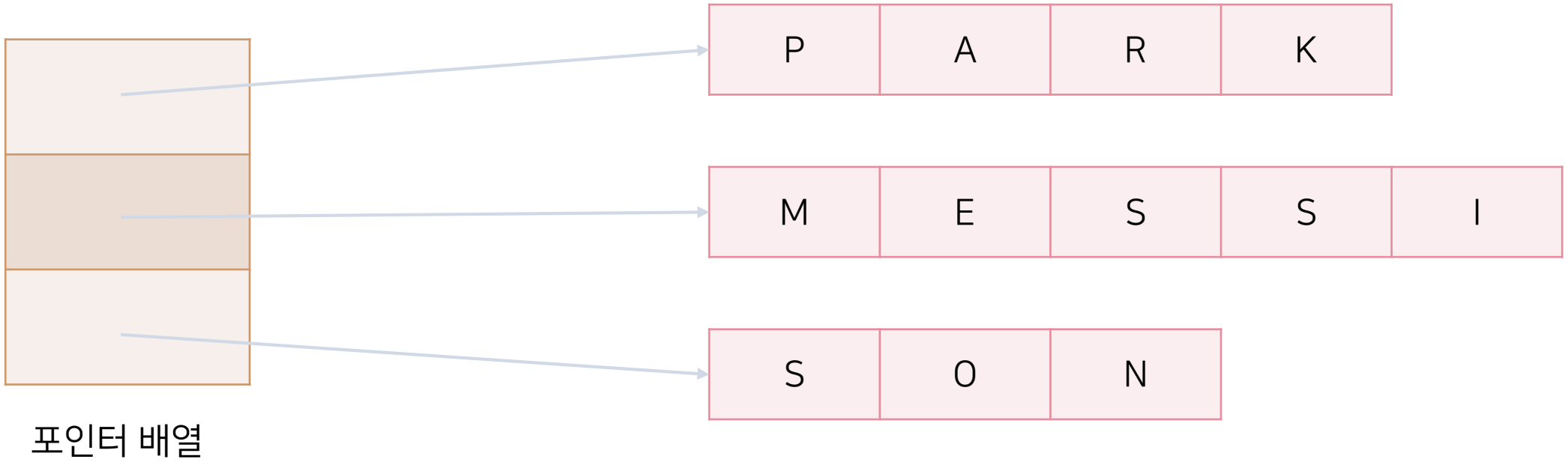
3 문자열의 2차배열

어떻게 하면 메모리에 낭비되는 공간과 제한 없이 구조를 만들 수 있을까요

P	A	R	K	₩0	
M	E	S	S	I	₩0
S	O	N	₩0		

3 문자열의 포인터 배열

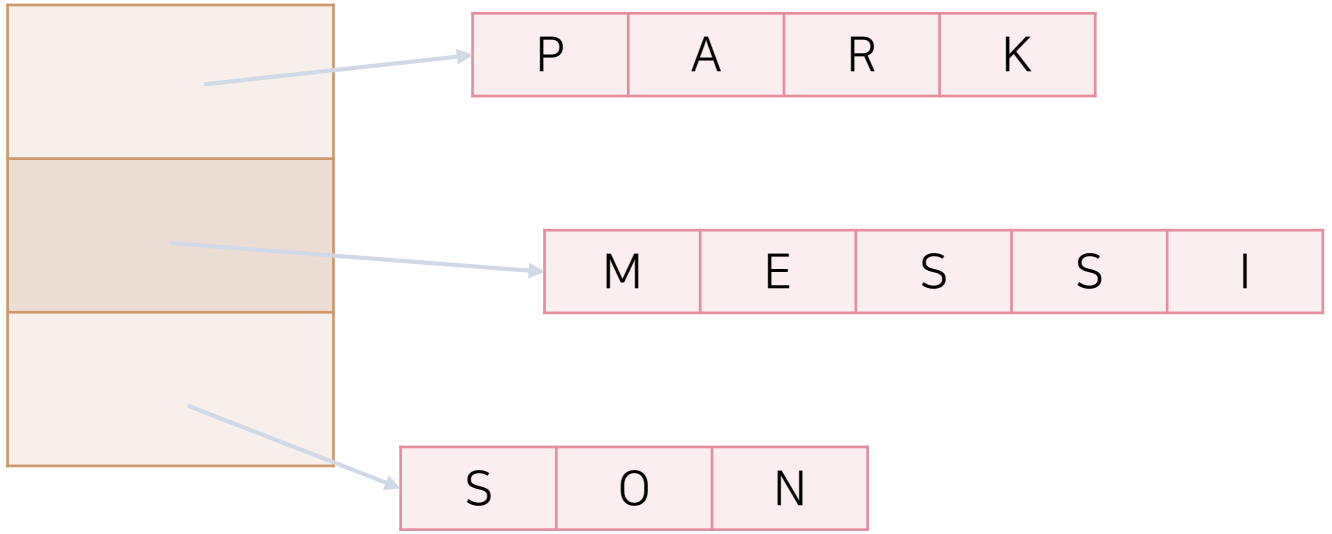
포인터로 이루어진 배열 즉 포인터 배열을 만들면 공간을 효율적으로 사용할수 있습니다.



3 문자열의 포인터 배열

아래의 names 는 포인터가 3개 들어있는 배열입니다.

```
int main(void) {
    char* names[3];
    names[0] = "PARK";
    names[1] = "MESSI";
    names[2] = "SON";
}
```

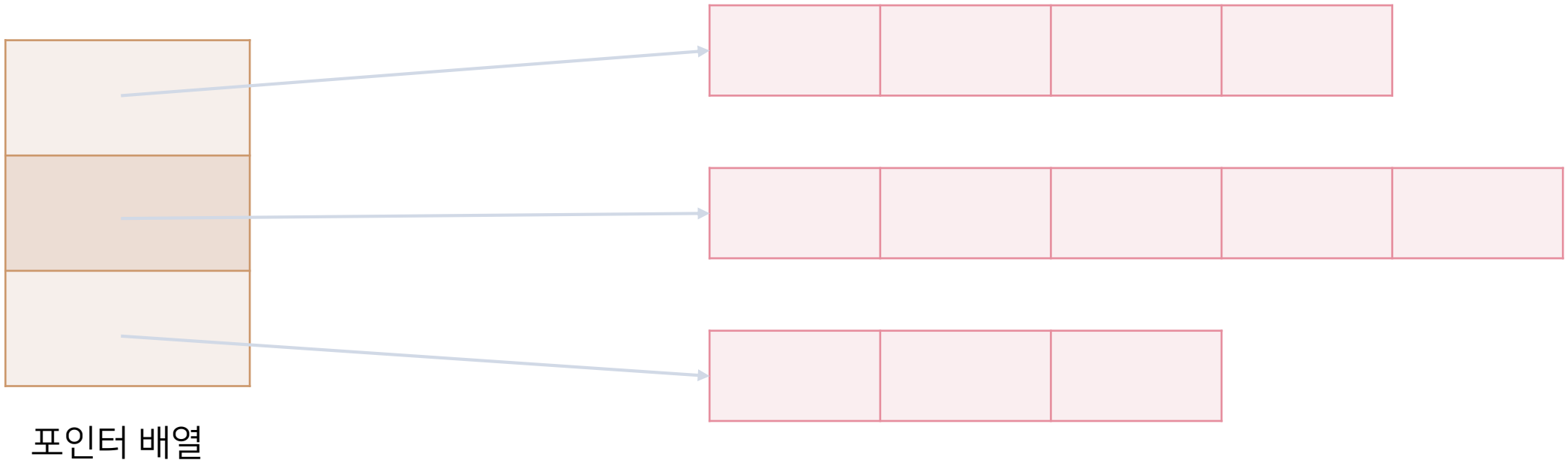


포인터 배열

3 문자열의 포인터 배열

각 줄의 데이터 크기가 제각각 인 경우 2차원 배열보다

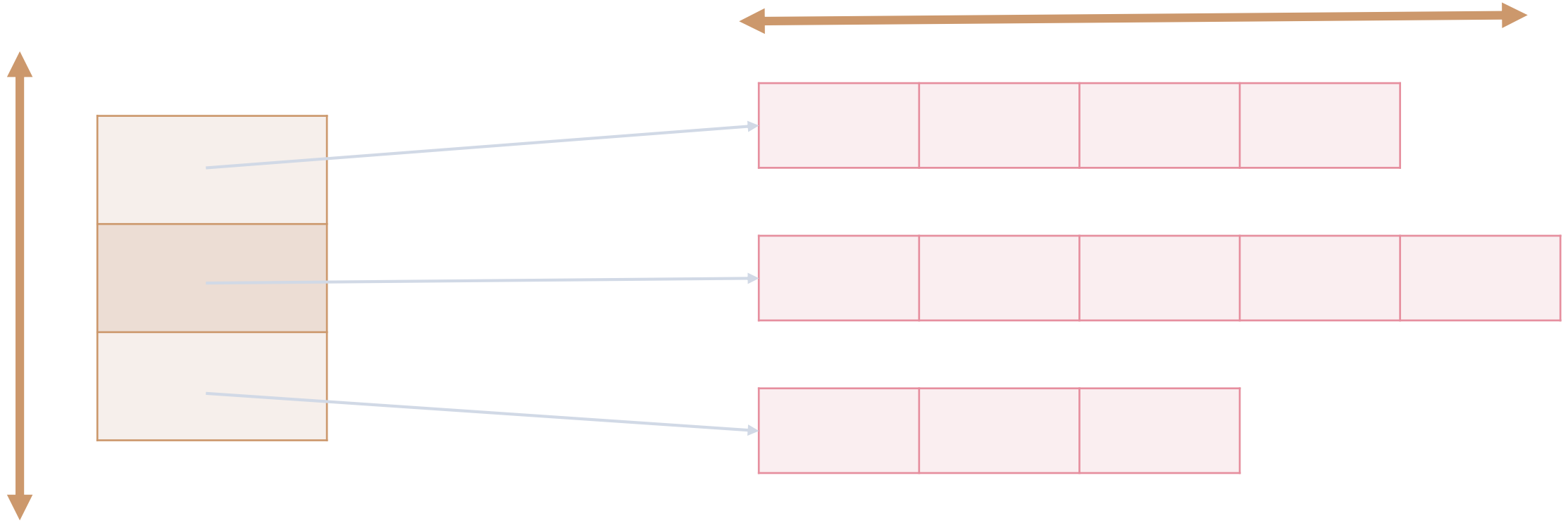
이러한 포인터 배열이 더 자주 사용됩니다. (문자열외에도 어떤 데이터라도 가능)



자료구조와 포인터

지금은 포인터가 배열에 저장되므로 크기가 고정되어 있지만

뒤에서 배울 동적메모리를 사용하면 크기를 자유롭게 지정할 수 있습니다.

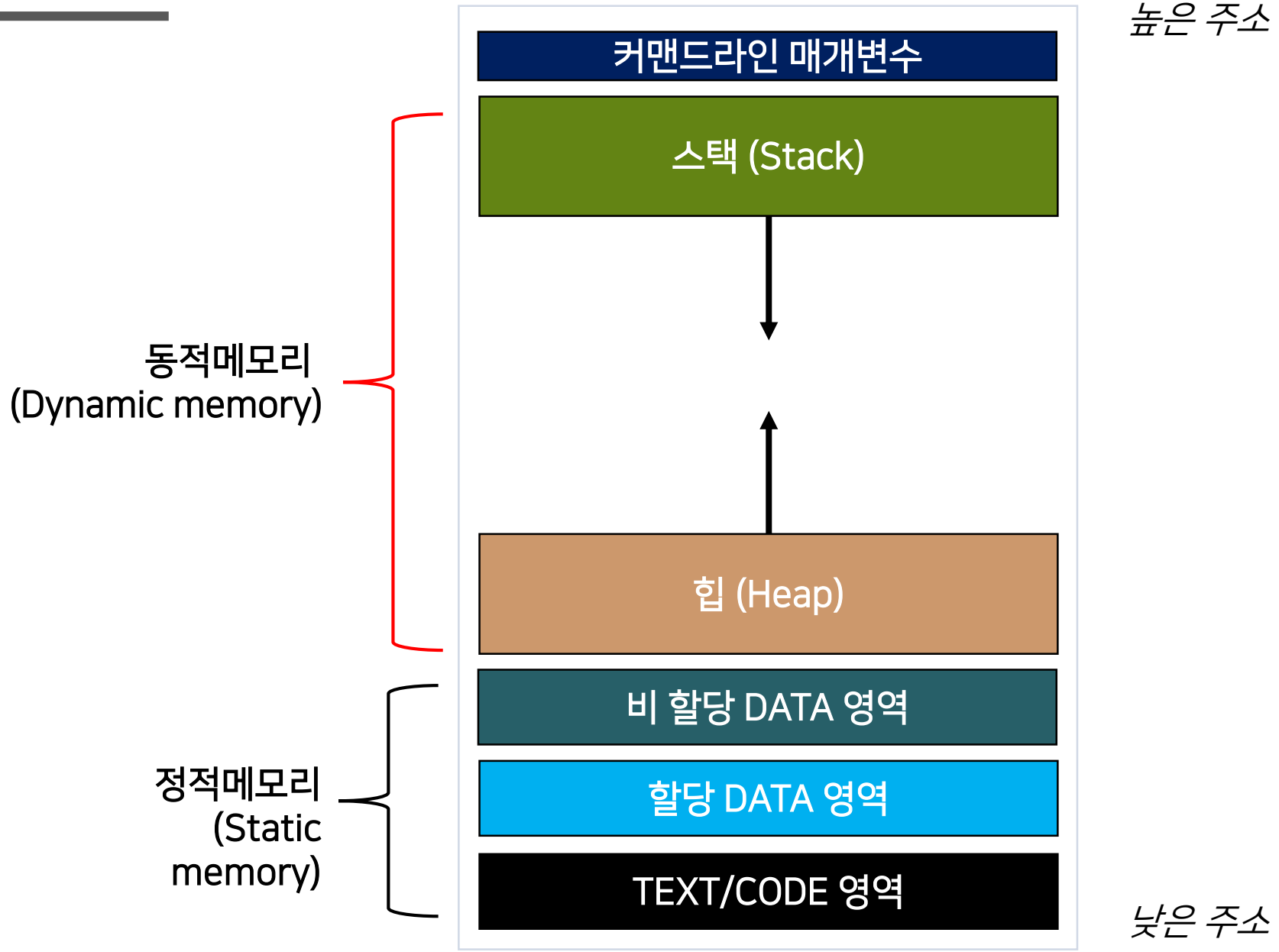


4.

동적 메모리 할당


IVE THE LIFE YOU'VE
LWAYS DREAMED OF.
BE FEARLESS IN THE
FACE OF ADVERSITY.
EVER STOP LEARNING.
SE YOUR IMAGINATION
WHENEVER POSSIBLE.
ECOGNIZE THE BE AUTY
THAT SURROUN
REMEMBER WHI
AME FROM, BU
LOSE SIGHT OF
YOU ARE GO

4 메모리 영역


높은 주소
커맨드라인 매개변수
스택 (Stack)
동적메모리
(Dynamic memory)
힙 (Heap)
비 할당 DATA 영역
정적메모리
(Static
memory)
할당 DATA 영역
TEXT/CODE 영역
낮은 주소

4 동적 메모리 할당

원하는 크기만큼 메모리를 할당하고 시작 주소를 받을 수 있습니다.

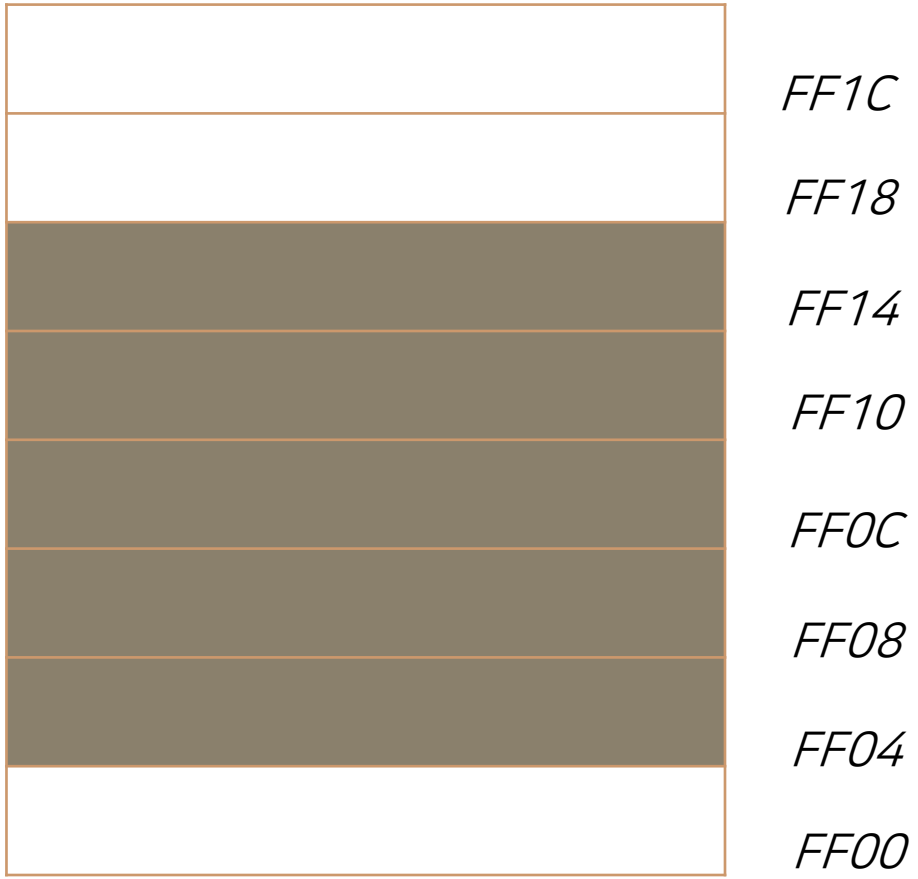
malloc (확보할 데이터크기)

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

int main(void) {

    int* p;
    p = (int*)malloc(sizeof(int) * 5);

    free(p);
}
```

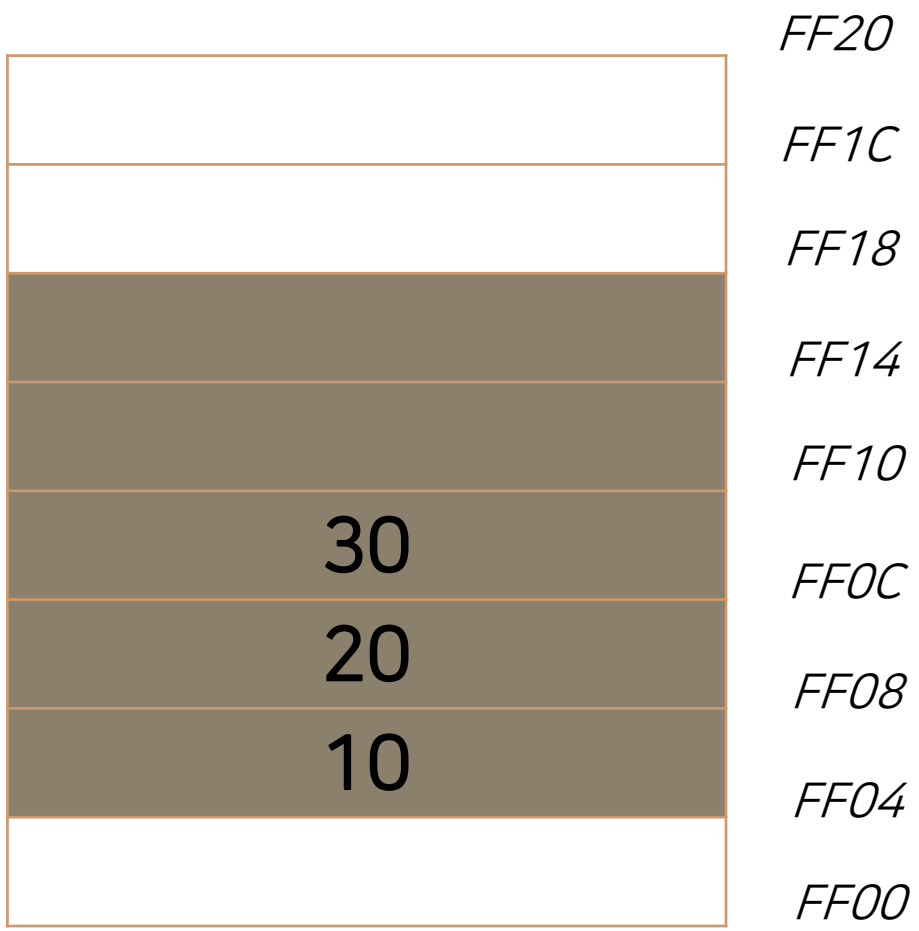


동적 메모리 할당

이렇게 만든 메모리 영역에 자유롭게 값을 읽고 쓸 수 있습니다.

사용이 끝난 메모리는 반드시 free 함수로 반환 해야합니다.

```
int main(void){
    int* p;
    p = (int*)malloc(sizeof(int) * 5);
    p[0] = 10;
    p[1] = 20;
    p[2] = 30;
    free(p);
}
```

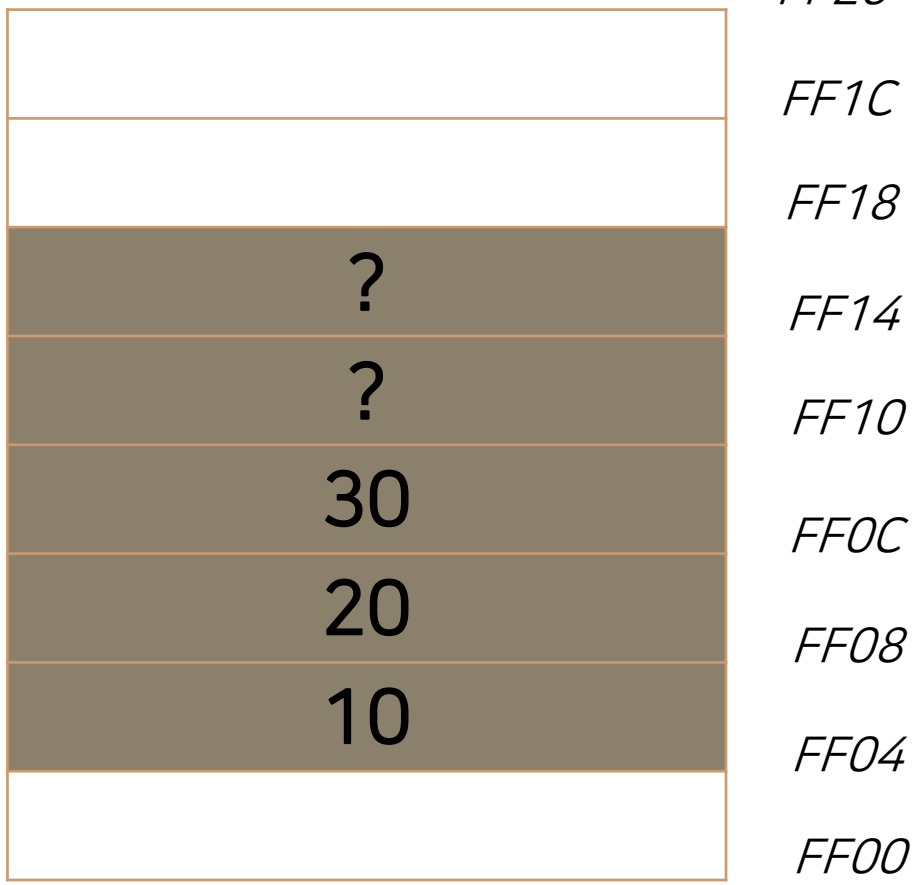


동적 메모리 할당

이 때 값을 넣지않은 메모리 안에 데이터는

어떤값이 될지 알 수 없습니다. (쓰레기값)

```c
int main(void){
    int* p;
    p = (int*)malloc(sizeof(int) * 5);
    p[0] = 10;
    p[1] = 20;
    p[2] = 30;
    printf("%d", p[3]);
    free(p);
}
```

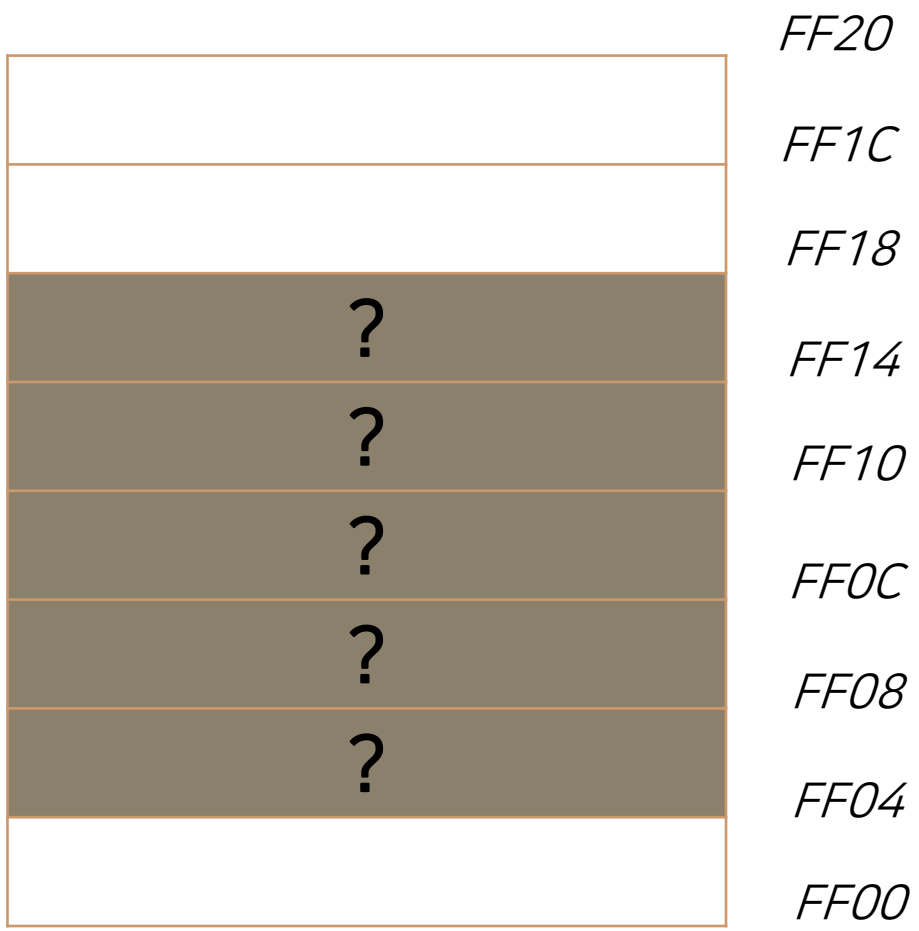


동적 메모리 할당

메모리를 할당해 줄 때 0으로 초기화 시켜주는 calloc 함수도 있습니다.

calloc (데이터개수, 데이터 하나의 크기)

```
int main(void){
    int* p;
    p = (int*)calloc(5, sizeof(int));
    p[0] = 10;
    p[1] = 20;
    p[2] = 30;
    printf("%d", p[3]);
    free(p);
}
```

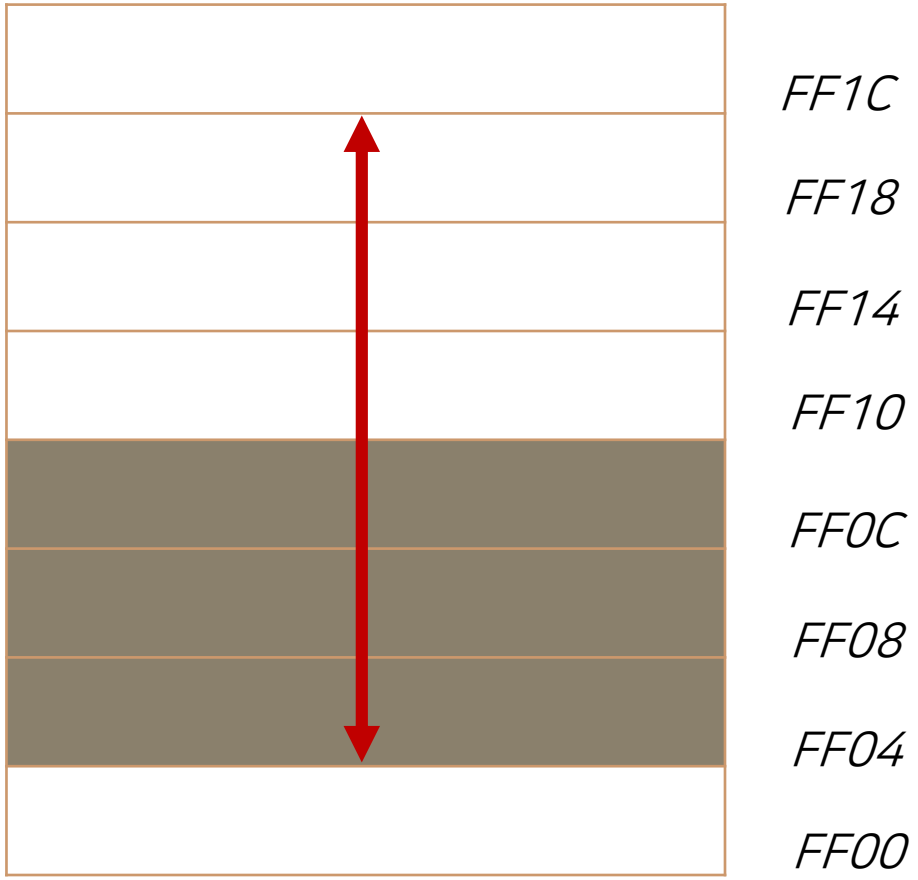


4 동적 메모리 할당

배열은 크기를 코드에 직접 적어야하는 반면에 동적메모리는 자유롭게 크기를 지정할 수 있습니다.

```c
int main(void){
    int* p, size, i;
    puts("저장할 숫자 개수를 입력:");
    scanf_s("%d", &size);
    p = (int*)malloc(sizeof(int) * size);

    for (i = 0; i < size; i++) {
        printf("%d번째 숫자입력:", i+1);
        scanf_s("%d", &p[i]);
    }
```



동적 메모리 할당

입력할 숫자를 정해
저장하는 전체코드

```c
#include <stdio.h>
#include <stdlib.h> // malloc()

int main(void){
    int* p, size, i;
    puts("저장할 숫자 개수를 입력:");
    scanf_s("%d", &size);
    p = (int*)malloc(sizeof(int) * size);

    for (i = 0; i < size; i++) {
        printf("%d번째 숫자입력:", i+1);
        scanf_s("%d", &p[i]);
    }
    for (i = 0; i < size; i++) {
        printf("%d번째 숫자 : %d₩n", i+1, p[i]);
    }
    free(p);
}
```

문자열 동적할당

문자열을 scanf_s 로 저장하는 코드를 작성해 보겠습니다.

```
int main(void){
    char name[10];
    scanf_s("%s", name, 10);
    printf("%s₩n", name);
}
```

k	i	m	₩0						

문자열 동적할당

문자열의 글자 크기를 확인하는 strlen 함수입니다. (₩0 제외)

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

int main(void){
    char name[10];
    scanf_s("%s", name, 10);
    int size = strlen(name);
    printf("%d₩n", size);
}
```

문자열 동적할당

딱 필요한 크기만큼 메모리를 할당해 보겠습니다.

```c
#include <stdio.h>
#include <string.h> // strlen()
#include <stdlib.h> // malloc()
int main(void){
    char name[10];
    scanf_s("%s", name, 10);
    int size = strlen(name);
    char* c = (char*)malloc(sizeof(char) * (size + 1));
}
```


문자열 동적할당

```c
#include <stdio.h>
#include <string.h> // strlen(), strcpy_s()
#include <stdlib.h> // malloc()
int main(void){
    char name[10];
    scanf_s("%s", name, 10);
    int size = strlen(name);
    char* c = (char*)malloc(sizeof(char) * (size + 1));
    strcpy_s(c, size + 1, name);
    printf("%s", c);
}
```

필요한 크기만큼 새로 만들어진 영역으로 strcpy_s 함수를 이용해 복사합니다.

4 문자열 동적할당 strdup() 함수

한번에 malloc 과 strcpy_s 를 동시에 해주는 함수가 있는데 바로 _strdup()함수 입니다.

```c
#include <string.h> // _strdup()

int main(void){
    char buffer[100];
    scanf_s("%s", buffer, 100);
    char* c = _strdup(buffer);
    printf("%s", c);
    free(c);
}
```