Chapter 4 Pointers and Arrays

Array and pointer의 관계

array의 이름은 pointer가 아니다!! array의 이름은 가끔 pointer로 사용될 수 있지만 이 둘은 같은 개념이 전혀 아니다!

Array란??

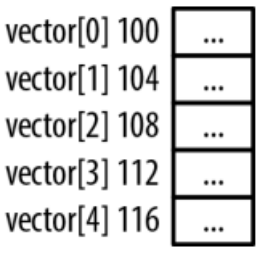
정의 : "같은 자료형"의 원소가 메모리에 "연속적으로" 존재하는 것.

📌2차원 배열도 Array는 같은 자료형의 원소가 메모리에 연속적으로 존재한다. pointer의 배열이랑은 약간 다른 느낌이다!!

1차원 배열

```
int vector[5];
```

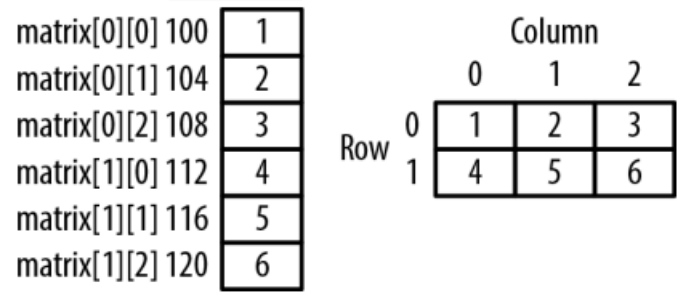
- 배열의 크기는 `컴파일 시점`에 결정된다.
- 배열의 사이즈는 `고정되어 있다`
- len을 구하는데, sizeof(vector) / sizeof(int)연산자를 사용할 수 있다.
- 원소의 주소가 `연속적으로` 위치한다.



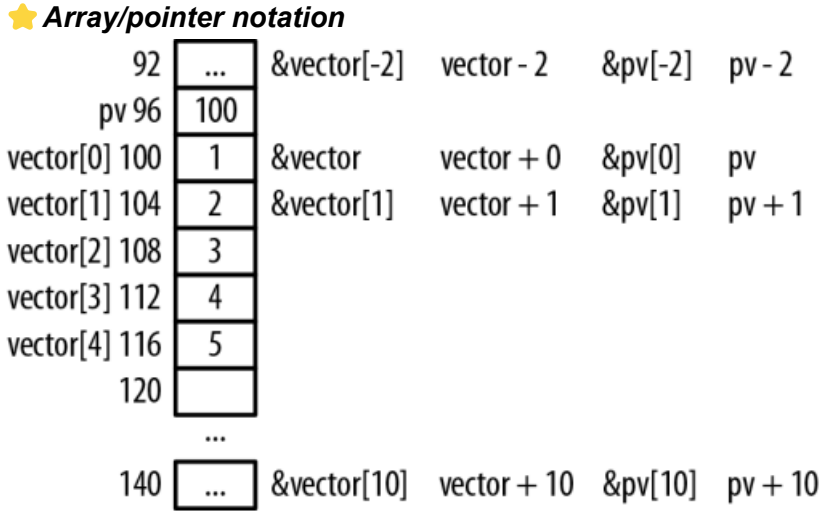
2차원 배열

```
int matrix[2][3] = {{1,2,3}, {4,5,6}};
```

- 배열의 크기는 `컴파일 시점`에 결정된다.
- 배열의 사이즈는 `고정되어 있다`
- len을 구하는데, sizeof(matrix) / sizeof(matrix[0])이 된다.
- 원소의 주소가 `연속적으로` 위치한다.



⭐ ***Array/pointer notation***



Differences Between Arrays and Pointers

```
int vector[5] = {1,2,3,4,5};
int *pv = vector;
```

Case 1) vector[i] && vector + i

`vector[i]`

- **표현**: `vector[i]`
- **의미**: 배열 `vector`의 `i`번째 요소에 접근한다는 뜻입니다.
- **작동 방식**:
  1. `vector`의 시작 주소에서 시작합니다.
  2. `i` 위치로 이동합니다: `i * sizeof(요소 타입)`.
  3. 해당 위치의 내용을 사용합니다.
- **생성 코드**: 컴파일러는 직접 `i`번째 요소의 주소를 계산하여 접근합니다.

`vector + i`

- **표현**: `*(vector + i)`
- **의미**: `vector` 시작점에서 `i` 요소만큼 떨어진 주소를 계산하고 그 위치의 내용을 사용합니다.
- **작동 방식**:
  1. 주소 계산: `vector의 시작 주소 + i * sizeof(요소 타입)`.
  2. 계산된 주소의 내용을 접근합니다.
- **생성 코드**: 컴파일러는 새로운 주소를 계산한 후 이를 역참조합니다

Case 2) sizeof연산자

`vector`

- sizeof(vector)를 사용하면 vector의 총 크기를 알 수 있다.
- sizeof(vector) / sizeof(int) 를 하면 길이를 알 수 있다.

`ptr`

- sizeof(ptr)연산을 하면 포인터 변수의 크기가 return된다.

Case3) rvalue && lvalue

1. pointer변수 `ptr`은 `lvalue`이다. 즉 특정 value를 할당할 수 있다.
2. vector는 `rvalue`이다. 특정 value를 할당할 수 없다.

```
ptr = ptr + 1;
vector = vector + 1; syntax error
```

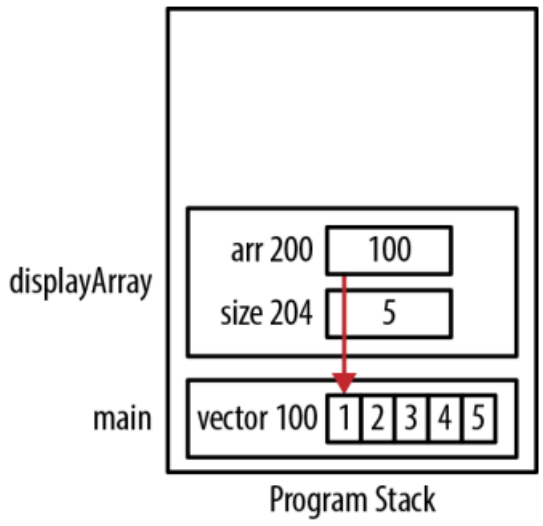
Passing a One-Dimension Array

- One-Dimension Array가 function에 pass되는 경우, array의 주소가 value로 인해 전해진다.
- 모든 array가 passing되지 않기 때문에 효율적으로 passing을 할 수 있다.
- function안에서는 배열의 크기를 알 수 있는 방법이 없기 때문에 size를 같이 넣어줘야한다.

Example Code

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

        int vector[5] = {1,2,3,4,5};
        displayArray(vector, 5);
///////

#include <stdio.h>

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

```
int main()
{
    int arr[] = {1,2,3,4,5};
    show_vector(arr, sizeof(arr)/sizeof(int));
    return (0);
}
```



Pointers and Multidimensional Arrays

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

void show_address(int matrix[][2])
{
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 2; j++)
            printf("%d ", &matrix[i][j]);
        printf("\n");
    }
}

void show_address_2(int **matrix)
{
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 2; j++)
            printf("%d ", &matrix[i][j]);
        printf("\n");
    }
}

int main()
{
    int con_matrix[2][2] = {{1,2}, {3,4}};
    int **matrix = (int**)malloc(sizeof(int*) * 2);
    for (int i = 0; i < 2; i++)
        matrix[i] = (int*)malloc(sizeof(int) * 2);
    show_address(con_matrix);
    printf("\n\n");
    show_address_2(matrix);
    return (0);
}
결과
1186985792 1186985796
1186985800 1186985804


194712592 194712596
194712624 194712628
```

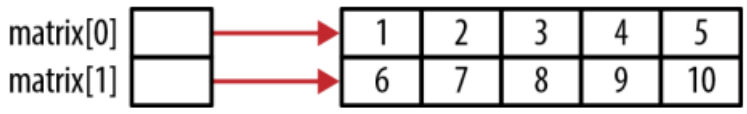
Case 1) 2차원 배열이 메모리상 연속적으로 있는 경우

```
int matrix[2][5] = {{1,2,3,4,5}, {6,7,8,9,10}};

int (*pmatrix)[5] = matrix; // pointer value
```

```
matrix[0][0]  Address: 100  Value: 1
matrix[0][1]  Address: 104  Value: 2
matrix[0][2]  Address: 108  Value: 3
matrix[0][3]  Address: 112  Value: 4
matrix[0][4]  Address: 116  Value: 5
matrix[1][0]  Address: 120  Value: 6
matrix[1][1]  Address: 124  Value: 7
matrix[1][2]  Address: 128  Value: 8
matrix[1][3]  Address: 132  Value: 9
matrix[1][4]  Address: 136  Value: 10
```

int*(pmatrix)[5]

- int[5] 형식의 배열에 대한 포인터
- matrix + 1을하면 offset은 20씩 올라간다.
- sizeof(matrix[0])을 하면 20이 나온다.

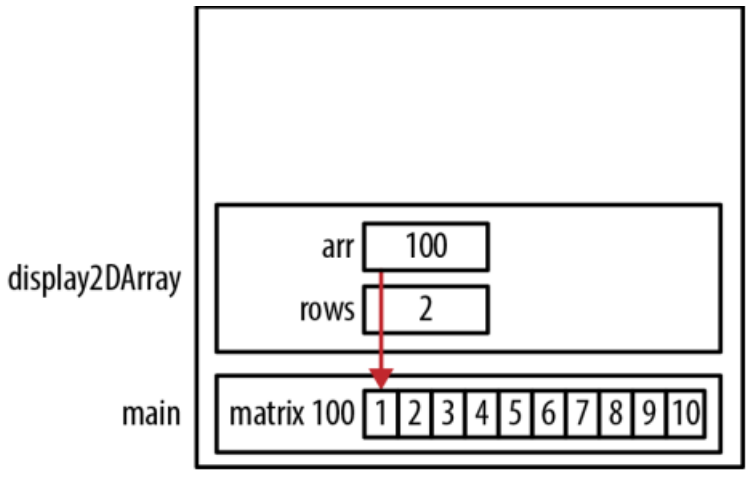


Passing a Multidimension Array

- Multidimension array를 함수 인자로 전달하기 위해서는 `array notaion`을 사용할 지, `pointer notation`을 사용할 지 정해야한다.
- array의 shape을 어떻게 전달할 지도 정해야한다. `shape이란, dimension의 크기와 숫자를 의미한다.`
- array notation을 사용하기 위해서는 array의 shape을 정해주지 않는다면 compiler가 오류를 발생시킬 수 있다.

```c
void display2DArray(int arr[][5], int rows)
// array notation

void display2DArray(int (*arr)[5], int rows)
// pointer notation
```

📌 `arr[]`는 포인터의 `implicit decalration`이고 `(*arr)`는 `explicit declaration`이다.



---

display2DArrayUnknownSize ⭐

```c
void display2DArrayUnknownSize(int *arr, int rows, int cols)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
            printf("%d ", *(arr + (i*cols) + j));
        printf("\n");
    }
}
```

- `printf`함수에서는 arr의 시작 주소에서 row와 col을 더해가면서 주소를 계산한다.

```c
display2DArrayUnknownSize(&matrix[0][0], 2, 5);
```

- 위의 방식으로 함수 호출이 가능하다.

```c
printf("%d ", arr[i][j]);
```

- 위의 방식으로는 함수 호출이 불가능하다.

- pointer는 이차원 배열으로 선언되지 않았기 때문이다 `int (*)[5]`
- 컴파일러는 dimension의 size를 모르기 때문에 오류를 발생시키게 된다.

```
printf ("%d ", (arr+i)[j]);
```

- 위의 방식으로는 함수 호출이 가능하다.
- `&matrix[0][0]` 으로 호출이 가능하다. `matrix` 으로는 호출이 불가능하다.
- `&matrix[0][0]` 은 정수형의 포인터이고, `matrix` 는 정수의 배열의 포인터이다.

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

void showMatrix(int *arr, int rows, int cols)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
            printf("%d ", *(arr + (i*cols) + j));
        printf("\n");
    }
}

void showMatrix2(int *arr, int rows, int cols)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
            printf("%d ", (arr + i)[j]);
        printf("\n");
    }
}


int main(void)
{
    int matrix[2][5] = {{1,2,3}, {4,5,6}};
    showMatrix(&matrix[0][0], 2, 5); // possible
    //showMatrix(matrix, 2, 5 );  impossible -> matrix는 int (*)[5]
    showMatrix2(&matrix[0][0], 2, 5);
    return (0);
}
```

Dynamically Allocating a Two-Dimension Array

2차원 배열을 동적으로 할당하기 위해서는 몇가지의 문제가 발생할 수 있다

1. array의 element가 연속적이지 않을 수 있다.
2. jagged array(각 행의 길이가 다른 배열이 발생할 수 있다)

📌

```
int matrix[2][5] = {{1,2,3,4,5},{6,7,8,9,10}};
```

- 위의 case에는 메모리들이 연속적으로 할당되어 있다.
- 그러나 `malloc` 을 사용해서 2차원 배열을 할당하는 방식에서는 `inner array` 들은 메모리에 연속적으로 있지 않다.

⭐ malloc으로 연속적인 2차원 배열 만드는 방법

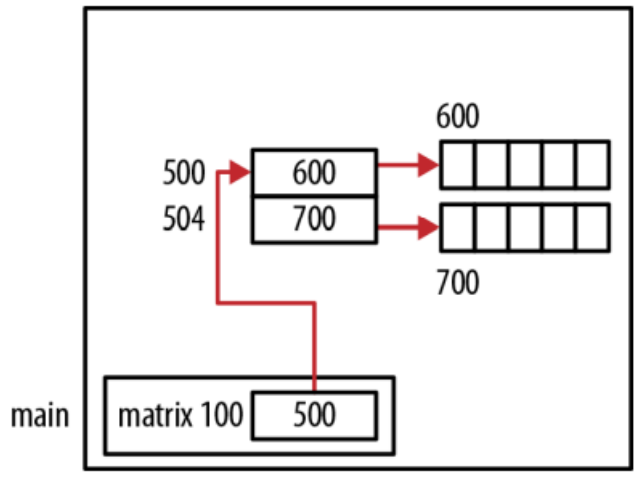
```
int *array = (int *)malloc(2 * 5 * sizeof(int));
```

- 접근방법 : array[i * 5 + j]\

Allocating Potentially Noncontiguous Memory

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

int** make_matrix() {
```

```
    int rows = 2;
    int cols = 5;

    int **matrix = (int**)malloc(sizeof(int*) * rows);
    for (int i = 0; i < rows; i++)
        matrix[i] = (int*)malloc(sizeof(int) * cols);
    return matrix;
}


void initialize_matrix(int **matrix) {
    int value = 1;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 5; j++) {
            matrix[i][j] = value++;
        }
     }
}

void show_matrix(int **matrix) {
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 5; j++)
            printf("%d ", matrix[i][j]);
        printf("\n");
    }
}


int main(void) {
    int **matrix = make_matrix();
    initialize_matrix(matrix);
    show_matrix(matrix);

    // 메모리 해제
    for (int i = 0; i < 2; i++) {
        free(matrix[i]);
    }
    free(matrix);
    return 0;
}
```



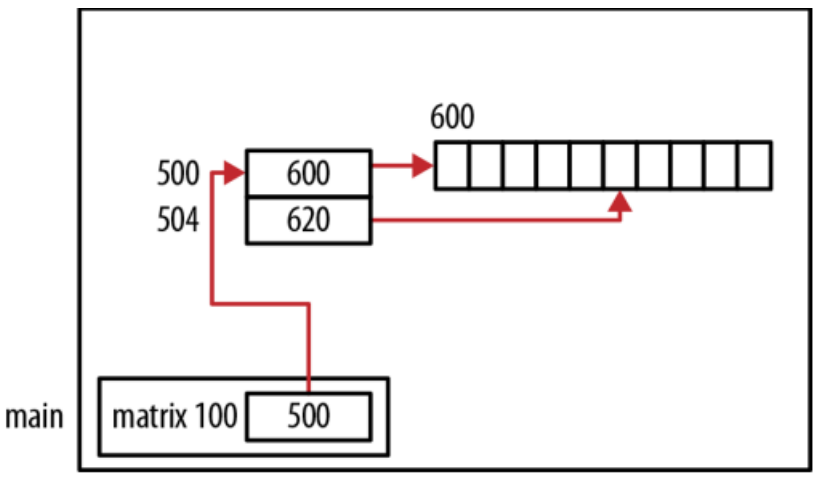
- 위의 방식은 `비연속적인` 방식으로 2차원 배열을 만드는 방법이다.
- `outer array`를 먼저 만든 뒤에, 각각의 `row`에서 malloc을해서 col을 만들어낸다.
- 각각의 array에서는 contiguous하다.

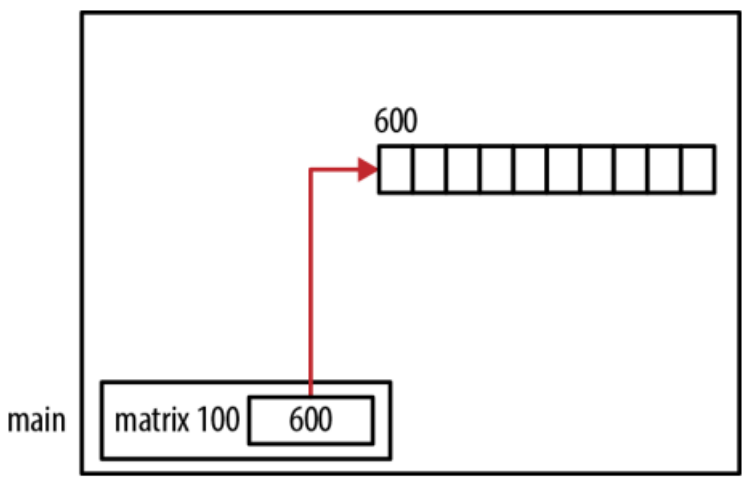
Allocating Contiguous Memory

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

int **make_contagious()
{
    int rows = 2;
    int cols = 5;

    int **matrix = (int**)malloc(rows * sizeof(int*));
    matrix[0] = (int*)malloc(rows * cols * sizeof(int));
    for(size_t i = 0; i < rows; i++)
        matrix[i] = matrix[0] + i * cols;
    return matrix;
}
```

```c
void init_matrix(int **matrix)
{
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 5; j++)
            matrix[i][j] = 1;
    }
}


void show_matrix(int **matrix)
{
    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 5; j++)
            printf("%d ", matrix[i][j]);
        printf("\n");
    }
}

int main(void)
{
    int **matrix = make_contagious();
    init_matrix(matrix);
    show_matrix(matrix);
    return (0);
}
```

- 포인터의 배열을 선언한 뒤에, 배열을 1번에 malloc하는 기술
- malloc을 하게 된다면, 메모리에 연속적으로 할당이 된다.





Jagged Array

Jagged Array : 2차원 배열이지만 행마다 다른 길이를 가지고 있는 배열을 의미한다.

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

int main() {
    int rows = 3;
    int **jaggedArray = (int **)malloc(rows * sizeof(int *));

    // 각 행의 다른 길이 할당
    jaggedArray[0] = (int *)malloc(2 * sizeof(int)); // 첫 번째 행
    jaggedArray[1] = (int *)malloc(3 * sizeof(int)); // 두 번째 행
    jaggedArray[2] = (int *)malloc(4 * sizeof(int)); // 세 번째 행

    // 값 할당 예시
    jaggedArray[0][0] = 1;
    jaggedArray[0][1] = 2;
    jaggedArray[1][0] = 3;
```

```
    jaggedArray[1][1] = 4;
    jaggedArray[1][2] = 5;
    jaggedArray[2][0] = 6;
    jaggedArray[2][1] = 7;
    jaggedArray[2][2] = 8;
    jaggedArray[2][3] = 9;
    // 출력
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < (i + 2); j++) {
            printf("%d ", jaggedArray[i][j]);
        }
        printf("\n");
    }
    // 메모리 해제
    for (int i = 0; i < rows; i++) {
        free(jaggedArray[i]);
    }
    free(jaggedArray);
    return 0;
}
```

문제점

1. 메모리 접근 복잡성

- **포인터 산술**: 재그드 배열은 다중 포인터를 사용하므로 메모리 접근이 포인터 연산에 의존합니다. 이로 인해 코드가 복잡해지고, 실수할 가능성이 높아집니다.
- **캐시 효율성**: 각 행이 비연속적으로 저장되기 때문에, 메모리 캐시 효율이 떨어질 수 있습니다. 연속적인 메모리 접근이 아니라면 캐시 미스가 발생할 수 있습니다.

2. 동적 할당의 오버헤드

- **메모리 관리**: 재그드 배열은 각 행에 대해 별도의 메모리 블록을 할당하므로, 메모리 할당과 해제 시 오버헤드가 증가할 수 있습니다.
- **해제의 복잡성**: 메모리 해제 시 각 행과 전체 배열을 따로 해제해야 하므로 코드가 복잡해질 수 있습니다.

3. 인덱스 계산

- **동적 크기 조정**: 각 행의 크기가 다르므로, 특정 위치를 계산할 때 인덱스 계산이 복잡해질 수 있습니다. 이러한 계산은 오류를 유발할 수 있습니다.

4. 성능 저하

- **메모리 분산**: 각 행이 별도의 메모리 블록에 할당되기 때문에 메모리가 분산됩니다. 이로 인해 데이터 접근 성능이 저하될 수 있습니다. 특히, 연속적인 데이터 접근이 필요한 경우 성능 저하가 두드러질 수 있습니다.