C programming - 6

임베디드스쿨2기

Lv1과정

2021. 05. 28

박태인

1. 배열 포인터

소스코드 분석
ㄴ 리눅스 노트북에 문제가 있어 Git hub에 올라와 있는 코드 내용으로 분석하였습니다.
ㄴ 분석 순서 : main 문 부터 시작해서 함수 사용시 해당 함수 내용 분석.

Main

```
int main(void)
{
        int len_A, len_B, len_U, len_V;
        int vectorA[3] = { 0 };
        int vectorB[3] = { 0 };
        int vecU[3] = { 1, -2, -1 };
        int vecV[3] = { -2, 0, -2 };
        int *vectorR;

        len_A = sizeof(vectorA) / sizeof(int);
        len_B = sizeof(vectorB) / sizeof(int);
        len_U = sizeof(vecU) / sizeof(int);
        len_V = sizeof(vecV) / sizeof(int);

        srand(time(NULL));

        // 배열의 이름은 배열의 대표 주소
        // 배열을 실제 메모리상에 순차적으로 배치됨
        // 배열의 대표주소에서 일정한 거리를
        // 지속적으로 이동한다면
        // 원소를 모두 순회할 수 있을 것이다.
        init_vector(vectorA, len_A);
        init_vector(vectorB, len_B);
```

- Len_A, B, U, V 의 길이 변수(상수)

- 벡터의 각 일차원 배열 선언
VectorA[3]
VectorB[3]
VecU[3]
VecV[3]

배열 포인터 int *vectorR

길이 변수 len은
Sizeof 함수를 활용한다.
Int형 변수이므로
Sizeof(int)로 배열의 길이를 구한다.

// **배열의 이름은 배열의 대표 주소**
// 배열은 실제 메모리 상에 순차적으로 배치됨
// 배열의 대표주소에서 일정한 거리를
// 지속적으로 이동한다면
// 원소를 모두 순회 할 수 있을 것이다. -> for문 활용

Init_vector함수가 등장한다.
인자로 vector 배열이름, 배열 길이를 넣는다.

1. 배열 포인터

Init_vector 함수

```
init_vector(vectorA, len_A);
init_vector(vectorB, len_B);
```

```
// 배열의 이름은 대표 주소 <<<
// 그러니까 포인터가
// 배열의 대표 주소를 인자로 받을 수 있다.
void init_vector(int *vector, int len)
{
        int i;

        for (i = 0; i < len; i++)
        {
                vector[i] = rand() % 8 + 1;
        }
}
```

// **배열의 이름은 대표 주소**
// 그러니까 포인터가
// 배열의 대표 주소를 인자로 받을 수 있다.

// 배열의 이름은 대표 주소! 니까 배열의 대표 주소를 인자로 받으려면
배열의 포인터 형 이어야 한다! **int ***

// 1~8까지의 랜덤한 수.
즉, 인자로 받은 배열의 길이 만큼 1~8의 랜덤한 수를 vector의 배열에 삽입.

Main

```
printf("vector A:\n");
print_vector(vectorA, len_A);
printf("vector B:\n");
print_vector(vectorB, len_B);
```

다시 Main으로 돌아와서

보면 새로운 함수 **print_vector** 을 볼 수 있다.

1. 배열 포인터

Print_vector 함수

```
void print_vector(int *vector, int len)
{
        int i;

        for (i = 0; i < len; i++)
        {
                printf("%3d", vector[i]);
        }

        printf("\n");
}
```

// 인자로 배열포인터, 배열 길이 상수

// 벡터배열의 값이 정해졌으니 그 값을 printf 하는 것.

Main

```
printf("vector A + B:\n");
vectorR = add_vector(vectorA, vectorB, len_A, len_B);
print_vector(vectorR, len_A);
```

// 자 슬슬 이제 벡터 합을 구할려고 폼 잡네요!
// **vectorR**이라는 곳에 벡터의 합 값을 넣은 심산 인듯 하다.
그러면 **add_vector의 반환형이 int *** 여야 할 것이고, 그것을 담는 변수인 vectorR 또한 마찬가지 일 것이다.
// A 배열의 길이만큼 vectorR을 출력하게 될 것이다.
// 이 말인 즉슨, add_vector 함수결과가 배열의 합이고 이 합의 배열을
// add_vector함수에서 구성하고 이놈을 가르키는게 vectorR 이므로
// **vectorR을 print_vector 하면 합을 한 배열을 print 하는 효과**가 나타난다.
// 여기서 길이를 len_A로 한건 그냥 한듯. 다 같은 길이라서.

1. 배열 포인터

Add_vector 함수

```
int *add_vector(int *vec_A, int *vec_B, int len_A, int len_B)
{
        int i;

        if (len_A != len_B)
        {
                printf("이 연산을 수행할 수 없습니다!\n");
                return NULL;
        }

        // malloc()
        // Stack | Heap | Data | Text
        // 이 중에서 malloc(), calloc()등의
        // 동적 할당을 수행하는 녀석들은
        // 모두 Heap 메모리에 할당된다.
        // 동적 할당이라 자유롭지만
        // 그만큼 성능은 떨어지게 된다.

        // 또한 기본 리턴 타입이 void * 이므로
        // 아래와 같이 사용하려는 데이터 타입에 맞게
        // 형 변환을 해줘야 하며
        // 사용하려는 바이트 수를
        // 입력 인자로 설정해줘야 한다.
        int *tmp = (int *)malloc(sizeof(int) * len_A);

        for (i = 0; i < len_A; i++)
        {
                tmp[i] = vec_A[i] + vec_B[i];
        }

        return tmp;
}
```

// 일단 두 벡터의 길이가 다르면 계산 불가.

//동적 할당인 **malloc을 왜 할까** ? 더한 배열을 새로운 영역에 만들어서 반환 시키기 위함이지!
// malloc( )
// Stack | Heap | Data | Text
// 의 메모리 영역에서 동적할당 하는 것들은 Heap에 해당됨.
// 동적할당이라 자유롭지만 그만큼 성능은 떨어지게 된다.

// 또한 기본 리턴 타입이 void * 이므로
// **아래와 같이 사용하려는 데이터 타입에 맞게**
// **형 변환을 해줘야 하며**
// 사용하려는 바이트 수를
// 입력 인자로 설정해 줘야 한다.

// 사용 바이트 수는 int와 그 길이 만큼.

// 그리고 생성된 tmp배열에 각각의 배열의 합을 삽입.

// int * 형의 tmp를 반환.

1. 배열 포인터

Main

```
printf("Inner Product:\n");
printf("두 개의 벡터가 서로 수직한가 ?\n");
printf("res = %f\n", dot_product(vecU, vecV, len_U, len_V));
printf("res = %f\n", dot_product(vectorA, vectorB, len_A, len_B));
```

// 이번에는 dot_product 라는 함수가 나타나내요.

dot_product 함수

```
// cos(90) = 0
// 내적의 특성을 통해 두 벡터가 서로 수직한지 확인한다.
// 각 원소간의 곱의 덧셈으로도 내적이 가능하며
// 여기서 0이 나온다는 뜻은 두 벡터가 수직함을 의미한다.
float dot_product(int *vec_A, int *vec_B, int len_A, int len_B)
{
        int i;
        float sum = 0;

        if (len_A != len_B)
        {
                printf("두 벡터의 연산은 불가능 하다\n");
                return -1;
        }

        for (i = 0; i < len_A; i++)
        {
                sum += vec_A[i] * vec_B[i];
        }

        return sum;
}
```

// cos(90) = 0
// 내적의 특성을 통해 두 벡터가 서로 수직한지 확인한다.
벡터의 수직을 알기 위한 것.
// **각 원소간의 곱의 덧셈으로도 내적이 가능**하며
// 여기서 0이 나온다는 뜻은 두 벡터가 수직함을 의미한다.

1. 배열 포인터

Main

```
// malloc()과 free()는 한 쌍에 해당한다.
// m이 할당이라면 f는 해제에 해당한다.
// 운영체제가 자동으로 하는 편이지만
// 빈번하게 발생하다보면 운영체제가 처리하지 못해
// 지속적으로 메모리 릭이 발생하게 되고 종극에 서버가 뻗어
// 회사의 손해가 막심해지게 된다.
free(vectorR);
```

// **malloc( )과 free( )는 한 쌍에 해당**한다.
// **m이 할당이라면 f는 해제에 해당**한다.
// 운영체제가 자동으로 하는 편 이지만
// 빈번하게 발생하다보면 운영체제가 처리하지 못해
// 지속적으로 메모리 릭(leak, 누수) 이 발생하게 되고 종극에 서버가 뻗어
// 회사의 손해가 막심해 진다.

2. 행열 구조체

구조체 선언

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

#define VEC_DIMENSION                3

typedef struct vector vec;                    // 구조체 vector을 typedef으로 vec로 정의

struct vector                                 // 구조체 int형 vector[3], int형 변수 len
{
        int vector[3];
        int len;
};
```

Main

```
int main(void)
{
        vec vecU;
        vec vecV;
        vec *vecR;                            // 구조체 포인터

        srand(time(NULL));

                                              // 주의 할 점은 배열처럼 배열 이름을 바로 하면 되는건 아니고,
                                              //   구조체의 주소를 넘겨 줘야 하기 때문에 &를 붙여 줘야 한다.
        init_vector(&vecU);
        init_vector(&vecV);                   // 마찬가지 구조체 V 넘길 때 & 붙임.
                                              // 즉 vecU와 vecV를 각각 만든 것임.
```

2. 행열 구조체

Init_vector 함수

```
void init_vector(vec *v)                // 인자를 구조체 vec의 포인터형, 배열처럼 구조체도 이름이 구조체의 대표주소!
{
        int i;

        for (i = 0; i < VEC_DIMENSION; i++)      // 3의 길이만큼 for문 돌리고
        {
                // 어떤 수를 7로 나눈 나머지는 ? 0 ~ 6        // 어떤 수를 7로 나눈 나머지는 ? 0 ~ 6
                // 위 범위 전체에 + 1을 하면 ? 1 ~ 7          // 위 범위 전체에서 +1을 하면 ? 1 ~ 7
                v->vector[i] = rand() % 7 + 1;              // 인자로 받은 구조체 v의 -> vector 배열에 1~7의 숫자로 랜덤으로 기입.
        }

        v->len = VEC_DIMENSION;         // 구조체 v의 int형 len 변수에 벡터치수 값 3 입력.
```

Main

```
        printf("vector U:\n");
        print_vector(vecU);         // vectorU의 값 출력.
        printf("vector V:\n");
        print_vector(vecV);         // vectorV의 값 출력.
```

2. 행열 구조체

print_vector 함수

```
void print_vector(vec v)                    // 여기서는 vec 포인터를 사용하지 않고, 그냥 vec형 사용.
{                                           // 만들어진 vec 구조체를 인자로.
        int i;

        for (i = 0; i < v.len; i++)         // 구조체의 len 값 까지 for문
        {
                printf("%3d", v.vector[i]);     // v구조체의 vector 배열
        }

        printf("\n");
}
```

Main

```
printf("vector U + V:\n");
vecR = add_vector(vecV, vecU);              // add_vector 함수가 vec * 반환형으로 vecR에게 값 전달.
print_vector(*vecR);                        // vecR은 vec* 형이고, print함수의 인자는 vec형 이므로 전달 할 때
                                            // vec형이 되도록 print함수 인자에 vecR 앞에 *를 붙여
                                            // 결국 넘어가는건 vec 형태로 넘어가도록 한다.
free(vecR);
                                    // 그리고 vecR이 malloc으로 동적할당 하므로 free 시켜 준다.
return 0;
```

2. 행열 구조체

Add_vector 함수

```
vec *add_vector(vec v, vec u)                    // 반환을 vec *형으로 한다는 거죠.
{
        int i;

        if (v.len != u.len)
        {
                printf("이 연산을 수행할 수 없습니다!\n");
                return NULL;
        }

        vec *tmp = (vec *)malloc(sizeof(vec));          // 구조체 크기 만큼 동적메모리 할당

        for (i = 0; i < v.len; i++)
        {
                tmp->vector[i] = v.vector[i] + u.vector[i];        // 아하.. ->는 포인터로 접근.
        }                                                          // 그냥 점은 포인터가 아닐시!!!! [쌤한테 물어본 포인트임]

        tmp->len = v.len;

        return tmp;
}
```

3. 행열 덧셈

Main

```
int main(void)
{
        int matA[2][2] = {
                { 1, 0 },
                { 0, 1 }
        };
        int matB[2][2] = {
                { 3, 3 },
                { 3, 3 }
        };

        // 이게 뭐지 ?
        // int *matR[2] -> int *    matR[2]
        // 위 코드와 아래 코드는 완전히 다른 것
        // int (*)[2]     matR;
        int (*matR)[2];

        printf("mat A:\n");
        print_mat(matA);
```

// 이건 뭐지??
// **int *matR[2] -> int * matR[2] : 요것은 1by 포인터 배열!**
// 위 코드와 아래 코드는 완전히 다른 것.
// **int (*)[2] matR; : 요것은 2by 배열 포인터!**

요것들의 차이는 뒤에
추가조사 해볼 것이다.

// int (*)[2] 형태의 반환형을 받기 위한 변수.

// print_mat 함수로 가보자.

3. 행열 덧셈

Print_mat 함수

```
int print_mat(int (*R)[2])
{
        int i, j;

        for (i = 0; i < 2; i++)
        {
                for (j = 0; j < 2; j++)
                {
                        printf("%3d", R[i][j]);
                }
                printf("\n");
        }
}
```

```
// 뭐 이런 문법이 다 있을까 ?
// C언어의 창시자인 켄 톰슨과 데니스 리치의 저서에 설명되어 있듯이
// C언어는 문법 자체에 포인터라는 특성 때문에
// 특수문자를 연달아 배치할 수 없다.
// 이로 인해 int[2] *를 int (*)[2] 형태로 표현해야 한다.
// 그러므로 원래라면
// int[2] *add_mat(int[2] *A, int[2] *B)를 아래와 같이 표기한다.
// int (*)[2] add_mat(int (*A)[2], int (*B)[2])
```

// 뭐 이런 문법이 다 있을까 ?
// C언어의 창시자인 켄 톰슨과 데니스 리치의 저서에 설명되어 있듯이
// **C언어는 문법 자체에 포인터라는 특성** 때문에
// 특수문자를 연달아 배치 할 수 없다.
// 이로 인해 **int[2] *를 int (*)[2] 형태로 표현해야 한다.**
-> int[2] 배열의 포인터인 int[2] *를 순서를 바꿔서 해야 하므로
-> int(*)[2] 인데, 의미는 int[2] 배열(2by 짜리 배열) 의 포인터라는 의미이다.

// 그러므로 원래라면
// int[2] *add_mat(int[2] *A, int[2] *B)를 아래와 같이 표기한다.
[2]와 *A는 자리를 그냥 바꾼 것.
// 즉, int (*A)[2] .. int[2] 배열의 포인터 A
// int (*)[2] add_mat(int (*A)[2], int (*B)[2]) -> **전체적으로 int[2]의*인데,**
add_mat 안에는 int[2]의* 인 A, int[2]*인 B가 인자로 들어가 있는 것이다.

추가적 이해

[질문]

선생님 사진의 함수의 인자 int (*R)[2] 는 int [2](*R)의 의미랑 같은 것으로,
int[2]..배열의 포인터 R 이라고 해석하면 될까요?
그리고 이러한 인자를 사용하는 이유가
예를 들어 인자로 int matA[2][2] 와 같은 (2 by 2) 의 행렬과 같은 배열을 받기 위함 이라면, 만약 int matA[2][3] 이나 int matA[3][3] 같은 경우의 행렬을 인자로 받게 된다면 int (*R)[3] 으로 인자를 받아야 하는 걸까요?

[답변]

넵. 맞습니다.

3. 행열 덧셈

Main

```
printf("mat B:\n");
print_mat(matB);

printf("mat A + B:\n");
matR = add_mat(matA, matB);        // 합의 값 계산
print_mat(matR);                   // 합의 값 출력

free(matR);
```

Add_mat 함수

```
// 배열을 리턴할 수 있는 고오급 기법
int (* add_mat(int (*A)[2], int (*B)[2]))[2]        // 반환형 int (*)[2] , 인자 (2by2) A행렬 + (2by2) B행렬
{
        int i, j;

        int (*tmp)[2] = (int (*)[2])malloc(sizeof(int) * 4);      // 2 by 2 이므로 int형이 4개인 크기.

        for (i = 0; i < 2; i++)
        {
                for (j = 0; j < 2; j++)
                {
                        tmp[i][j] = A[i][j] + B[i][j];
                }
        }

        return tmp;                // 합의 행렬 값 출력.
}
```

4. 구조체 버전 벡터 연산

구조체 선언

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

// typedef를 통해 struct test를 test로 축약시킴
// 길게 쓰기 싫어서 적는다고 봐도 된다.
// 사실 좀 더 깊은 의미가 있는데
// 자바의 인터페이스 역할도 할 수 있긴하다.
typedef struct test test;

struct address
{
        char *city;
        char *street;
        int zipcode;
};

// { 10,     주소      } ----------> { 20, 주소 } -> { 30, 주소 }
//  int, 구조체 포인터               이하 동문        이하 동문
```

// **typedef를 통해 struct test를 test로 축약시킴**
// 길게 쓰기 싫어서 적는다고 봐도 된다.
// 사실 좀 더 깊은 의미가 있는데
// 자바의 인터페이스 역할도 할 수 있긴하다.

// { 10, 주소 } ----------> { 20, 주소 } -> { 30, 주소 }
// int, 구조체 포인터 이하 동문 이하 동문

4. 구조체 버전 벡터 연산

구조체 선언

```
struct test
{
        int data;
        // 자료구조
        struct test *self_pointer;
        struct address addr;
        // 함수 포인터 - 다형성
        // 이 부분은 Lv3 소프트웨어 아키텍처링(설계) 파트에서 좀 더 상세하게 다룰 내용
        // 생산성을 증대시키면서 유지보수성을 함께 올릴 수 있는 방법은 무엇인가 ?
        // 100명 ~ 1000명 그 이상의 직원들이 함께 일을 한다면
        // 어떻게 해야 효율적이고 불필요한 시간 낭비를 절약할 수 있을것인가 ?
        // 이와 같은 문제에 대한 해결책이 사실 이 단순한 함수포인터에 녹아있다.
        // 레벨 2에서 소프트웨어 집중한 대규모 팀 프로젝트를 통해
        // 왜 이따구로 코딩을 하면 안되고 왜 문서를 요따구로 만들면 안되는지
        // 직접 부딪혀 보면서 파악하는 시간을 가지고
        // 이후 요 내용을 통해서 아 ~ 하 ? 하면 된다고 보시면 됩니다.

        // 숙제에서 함수 포인터는 패스해도 무방합니다.
        // 현재로서는 이 부분의 심도 있는 이해가 불가하기 때문
        // 어떤 흐름 때문에 사용한다는 파악하도록 한다.
        int *(* real)(int *, char ***, struct test *);
        void (* basic)(void);
};
```

// **함수 포인터 - 다형성(여러가지 형태가 존재)**
// 이부분은 Lv3 소프트웨어 아키텍처링(설계) 파트에서 좀 더 상세하게 다룰 내용
// 생산성을 증대시키면서 유지보수성을 함께 올릴 수 있는 방법은 무엇인가 ?
// 100명 ~ 1000명 그 이상의 직원들이 함께 일을 한다면
// 어떻게 해야 효율적이고 불필요한 시간 낭비를 절약 할 수 있을 것인가?
// 이와 같은 문제에 대한 해결책이 사실 이 단순한 함수포인터에 녹아 있다.
// 레벨 2에서 소프트웨어 집중한 대규모 팀 프로젝트를 통해
// 왜 이따구로 코딩을 하면 안되고 왜 문서를 요따구로 만들면 안되는지
// 직접 부딪혀 보면서 파악하는 시간을 가지고
// 이후 요 내용을 통해서 아 ~ 하 ? 하면 된다고 보면 된다.

// 숙제에서 함수 포인터는 패스해도 무방하다.
// 현재로서는 이 부분에서 심도 있는 이해가 불가하기 때문
// 어떤 흐름 때문에 사용한다는 파악하도록 한다.

// 이 부분은 잘 이해가 되진 않지만 일단 넘어가도록 한다.

4. 구조체 버전 벡터 연산

Main

```
int main(void)
{
        // 구조체를 사용하는 방법
        // 1. struct를 적는다.
        // 2. 구조체 이름을 작성한다.
        // 3. 구조체 내부에서 사용할 데이터 타입과
        //    필드명(변수 같은것이지만 변수는 아님)을 작성한다.
        // 4. 활용할 때 struct 구조체명 변수명 형식으로 작성한다.
        // 1 ~ 3번까지는 새로운 데이터타입을 만든것이다.
        // 4번이 실제 활용하는 방식이며 결국 이것도 변수다.
        // 중요한 것은 C언어에서 구조체는
        // 우리만의 전용 데이터타입을 만드는 방식이란 것이다.
        struct test sample;
        // int sample
        // float sample
        // double sample
        // char *sample
        // char **sample
        // int *****sample
        // double **************************sample
        test sample2;

        srand(time(NULL));

        init_test_struct(&sample, NULL);
        print_test_struct(sample);
        printf("\n");
```

// **구조체를 사용하는 방법**
// 1. struct를 적는다.
// 2. 구조체 이름을 작성한다.
// 3. 구조체 내부에서 사용할 데이터 타입과
// 필드명(변수 같은 것이지만 변수는 아님)을 작성한다.
// 4. 활용할 때 struct 구조체명 변수명 형식으로 작성한다.
// **1~3번까지는 새로운 데이터 타입을 만든 것이다.**
// **4번이 실제 활용하는 방식이며 결국 이것도 변수다.**
// **중요한 것은 C언어에서 구조체는**
// **우리만의 전용 데이터타입을 만드는 방식이란 것이다!!!**

4. 구조체 버전 벡터 연산

Init_test_struct 함수

```
void init_test_struct(struct test *sample, test *link)
{
        // -> 연산자: 간접 참조 연산으로 구조체 연산에만 활용됨
        // 구조체가 포인터로 주어졌을 경우 해당 포인터를 통해 구조체 필드에 접근한다면
        // -> 화살표 연산자를 사용해야함

        // . 연산자: 직접 참조 연산으로 마찬가지로 구조체에서 활용됨
        // 구조체가 포인터가 아닌 형태로 주어졌을 경우 사용함
        // 구조체 내부의 멤버 필드에 접근하기 위해 사용됨
        sample->data = rand() % 10 + 1;
        sample->self_pointer = link;

        char city[] = "Seoul";
        char street[] = "마포구 큰우물로 76";

        // 포인터 변수는 메모리를 할당해야 사용할 수 있다.
        // Heap 메모리는 일반 Stack이나 다른 메모리들과 다르게
        // 여러 프로세스들이 구동되면서 서로 얽힐 수도 있는데
        // +1을 통해 NULL 문자를 할당해두면
        // 뒤에 있는 내용을 붙어서 읽지 않고 끝이 어디인지 정확하게 알 수 있음
        sample->addr.city = (char *)malloc(strlen(city) + 1);
        // 위의 메모리 할당 없이 strcpy만 하는 경우 오류
        // sample->addr.city에 위의 city 배열을 복사한다.
        // string copy의 약자: strcpy
        strcpy(sample->addr.city, city);

        sample->addr.street = (char *)malloc(strlen(street) + 1);
        strcpy(sample->addr.street, street);

        sample->addr.zipcode = 12345;

        sample->real = NULL;
        sample->basic = print_test;
}
```

// **-> 연산자** : 간접 참조(주소 접근) 연산으로 구조체 연산에만 활용됨
// 구조체가 포인터로 주어졌을 경우 해당 포인터를 통해 구조체 필드에 접근 한다면
// -> 화살표 연산자를 사용해야 함

// **. 연산자** : 직접 참조(값 접근) 연산으로 마찬가지로 구조체에서 활용됨
// 구조체가 포인터가 아닌 형태로 주어졌을 경우 사용함
// 구조체 내부의 멤버 필드에 접근하기 위해 사용됨.

// NULL을 넣는다면 모든 값 초기화. 셀프 포인터.

// **포인터 변수는 메모리 할당해야 사용 할 수 있다.**
// Heap 메모리는 일반 Stack이나 다른 메모리들과 다르게
// 여러 프로세스들이 구동되면서 서로 얽힐 수도 있는데
// +1을 통해 NULL 문자를 할당해두면
// 뒤에 있는 내용을 붙어서 읽지 않고 끝이 어디인지 정확하게 알 수 있다.

// 메모리 할당 없이 strcpy만 하는 경우 오류
// sample->addr.city에 (만든 동적할당 메모리에) 위의 city 배열을 복사한다.
// string copy의 약자 : strcpy

4. 구조체 버전 벡터 연산

Print_test_struct 함수

```
void print_test_struct(struct test sample)
{
        printf("data = %d\n", sample.data);
        printf("self_pointer = 0x%x\n", sample.self_pointer);
        printf("city = %s\n", sample.addr.city);
        printf("street = %s\n", sample.addr.street);
        printf("zipcode = %d\n", sample.addr.zipcode);
        printf("real = 0x%x\n", sample.real);
        // 함수의 이름도 결국 주소값이다.
        // 그러므로 포인터로 저장할 수 있다.
        // 이것이 함수 포인터다.
        printf("basic = 0x%x\n", sample.basic);
        printf("print_test = 0x%x\n", print_test);
}
```

// **함수의 이름도 결국 주소값이다**.
// 그러므로 포인터로 저장할 수 있다.
// 이것이 함수 포인터다.

4. 구조체 버전 벡터 연산

Main

```
init_test_struct(&sample2, &sample);          // 2번째 sample2 입력. sample 구조체 셀프 포인터
print_test_struct(sample2);
printf("sample address = 0x%x\n", &sample);

clear_malloc(sample);
clear_malloc(sample2);

return 0;
```

Clear_malloc 함수

```
void clear_malloc(test sample)
{
        free(sample.addr.city);              // 동적 할당 받은 구조체 값 free
        free(sample.addr.street);
}
```

5. (추가조사) 포인터 배열, 배열 포인터

```
// int *matR[2] -> int *     matR[2]   : 요것은 1by 포인터 배열!
// 위 코드와 아래 코드는 완전히 다른 것.
// int (*)[2]    matR;                  : 요것은 2by 배열 포인터!
```

ㄴ 앞서 행렬덧셈 코드 분석 중 위의 차이점을 살펴 보기 위한 페이지이다. (12 page 참조)

* 포인트 배열이란?

ㄴ 포인터 배열이란 말 그대로 포인터를 배로 나열해 놓은 것을 말합니다. 즉, **포인터 변수의 배열**이죠.
각각의 index에 여러 개의 포인터르 넣을 수 있습니다.

```
char* arr[5];
int* arr[5];
void* arr[5];
```

ㄴ 위와 같이 선언하여 사용합니다. **Char* arr[5]는 5개의 char형 포인터**를,
int* arr[5]는 int형 포인터 5개를 저장하고 있는 배열이 됩니다.
그냥 단순 주소값만을 저장하고 싶을 때는 주로 void로 선언하며 이렇게 선언할 경우
이후 어떤 타입이든지 변환이 가능합니다.

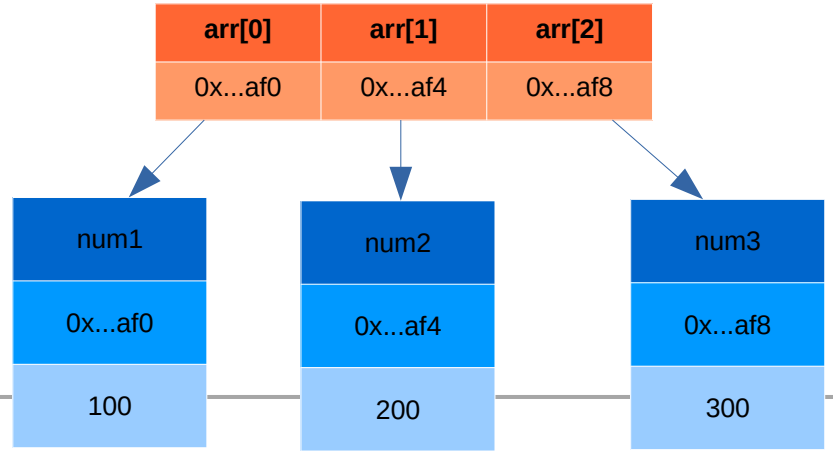
5. (추가조사) 포인터 배열, 배열 포인터

포인터배열 예제

```c
#include <stdio.h>

int main()
{
        int num1 = 100;
        int num2 = 200;
        int num3 = 300;

        int*arr[3] = {&num1, &num2, &num3};

        for(int i=0; i<3; i++)
        {
                printf("arr[%d] = 주소값 : %p / 값 :%d\n", i,arr[i], *arr[i]);
        }
}
```

```
arr[0] = 주소값 : 0x7ffd79cccaf0 / 값 :100
arr[1] = 주소값 : 0x7ffd79cccaf4 / 값 :200
arr[2] = 주소값 : 0x7ffd79cccaf8 / 값 :300
```

Int 형 포인터배열에 주소값을 넣어 놓고 *arr에서는 그 배열의 값을 출력하는 모습을 볼 수 있습니다.
단순히 포인터를 배열로 나열해 놓은 것에 불과 합니다.



왼쪽과 같이 포인터 배열의 index 마다 참조하는 주소값을 저장하여 활용할 수 있습니다.

5. (추가조사) 포인터 배열, 배열 포인터

* 배열 포인터란?

ㄴ 배열 포인터란 배열을 가르키는 포인터를 말합니다. 배열은 변수들을 메모리상에 일렬로 나열해놓은 것과 마찬가지 입니다. 배열도 메모리상에 존재하므로 엄연히 주소값이 존재합니다. **배열 포인터란 이 주소값을 가리키는 포인터**를 말합니다. 이것이 유용한 이유는 바로 2차원 이상의 배열을 가리킬 때 포인터를 통해 배열과 같은 인덱싱을 할 수 있기 때문 입니다. 함수에 2차원 이상의 배열을 파라미터로 던질 때 유용하게 사용 됩니다.

배열의 주소 값

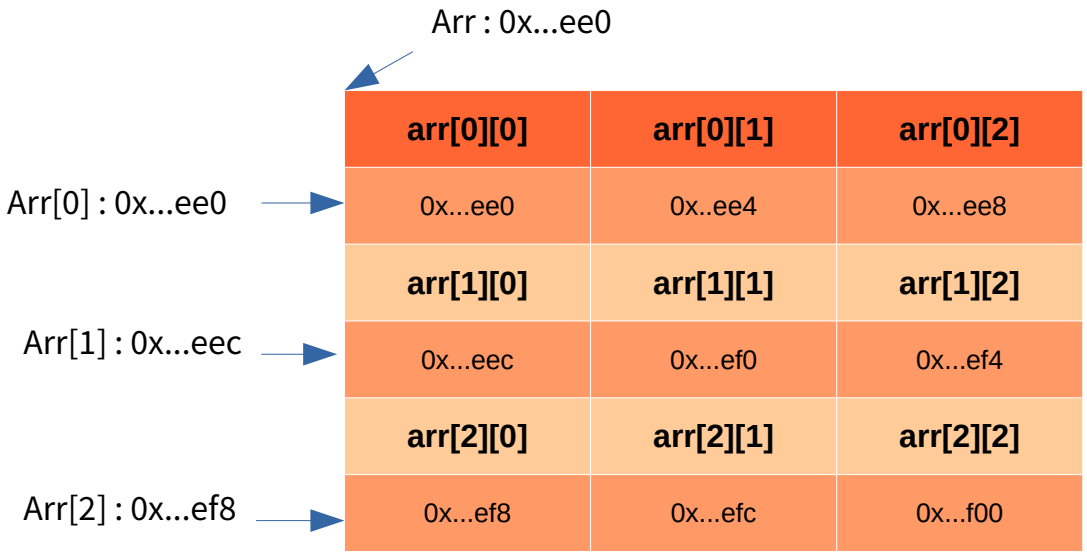
```c
#include <stdio.h>
int main() {
    int arr[3][3] = { {10, 20, 30}, {100, 200, 300}, {100, 200, 300}};

    printf("arr의 주소값 : %p\n", &arr);

    for (int i = 0; i < 3; i++) {
        printf("arr[%d]의 주소값 : %p\n",i, &arr[i]);

        for (int j = 0; j < 3; j++) {
            printf("arr[%d][%d]의 주소값 : %p / 값 :%d\n", i, j, &arr[i][j], arr[i][j]);
        }
    }
}
```

```
arr의 주소값 : 0x7ffce9815ee0
arr[0]의 주소값 : 0x7ffce9815ee0
arr[0][0]의 주소값 : 0x7ffce9815ee0 / 값 :10
arr[0][1]의 주소값 : 0x7ffce9815ee4 / 값 :20
arr[0][2]의 주소값 : 0x7ffce9815ee8 / 값 :30
arr[1]의 주소값 : 0x7ffce9815eec
arr[1][0]의 주소값 : 0x7ffce9815eec / 값 :100
arr[1][1]의 주소값 : 0x7ffce9815ef0 / 값 :200
arr[1][2]의 주소값 : 0x7ffce9815ef4 / 값 :300
arr[2]의 주소값 : 0x7ffce9815ef8
arr[2][0]의 주소값 : 0x7ffce9815ef8 / 값 :100
arr[2][1]의 주소값 : 0x7ffce9815efc / 값 :200
arr[2][2]의 주소값 : 0x7ffce9815f00 / 값 :300
```

위의 예제처럼 배열에 주소값을 찍어서 값을 확인 할 수 있습니다.
위의 예제에서 알 수 있듯 2차원배열 arr은 arr[0][0]의 주소와
같으며 int형 배열이므로 주소값이 4씩 증가하는 것을 확인 할 수 있습니다.

5. (추가조사) 포인터 배열, 배열 포인터

```
arr의 주소값 : 0x7ffce9815ee0
arr[0]의 주소값 : 0x7ffce9815ee0
arr[0][0]의 주소값 : 0x7ffce9815ee0 / 값 :10
arr[0][1]의 주소값 : 0x7ffce9815ee4 / 값 :20
arr[0][2]의 주소값 : 0x7ffce9815ee8 / 값 :30
arr[1]의 주소값 : 0x7ffce9815eec
arr[1][0]의 주소값 : 0x7ffce9815eec / 값 :100
arr[1][1]의 주소값 : 0x7ffce9815ef0 / 값 :200
arr[1][2]의 주소값 : 0x7ffce9815ef4 / 값 :300
arr[2]의 주소값 : 0x7ffce9815ef8
arr[2][0]의 주소값 : 0x7ffce9815ef8 / 값 :100
arr[2][1]의 주소값 : 0x7ffce9815efc / 값 :200
arr[2][2]의 주소값 : 0x7ffce9815f00 / 값 :300
```



위는 앞서 나타난 값을 그림으로 표현해 보았습니다.
2차원 배열을 선언하여 주소값을 확인해 보면 **1차원 배열의 주소는 2차원 배열의 [x][0]번째 주소를 가리킨다는 것을 알 수 있고, 2차원 배열의 주소는 2차원 배열의 [0][0]의 주소를 가리킨다는 것을 확인 할 수 있습니다**.
이러한 성질을 활용하여 배열 포인터를 다양한 곳에서 활용 할 수 있습니다.

5. (추가조사) 포인터 배열, 배열 포인터

배열 포인터 활용 예제 (2차원 배열을 함수의 파라미터로 보낼 때)

```c
#include <stdio.h>
void change_array(int(*arr)[3]) { //배열 포인터
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            arr[i][j] = 0; //arr의 값 0으로 변경
        }
    }
}

int main() {
    int arr[3][3] = { {10, 20, 30}, {100, 200, 300}, {100, 200, 300}};

    change_array(arr);

    //출력
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("arr[%d][%d] : %d ", i, j, arr[i][j]);
        }
        printf("\n");
     }
}
```

```
arr[0][0] : 0 arr[0][1] : 0 arr[0][2] : 0
arr[1][0] : 0 arr[1][1] : 0 arr[1][2] : 0
arr[2][0] : 0 arr[2][1] : 0 arr[2][2] : 0
```

함수에서 다차원 배열을 파라미터로 받아야 할 때 배열 포인터르 활용 할 수 있습니다.
위와 같이 C 언어의 함수에서 **다차원 배열의 파라미터를 받을 때는**
위와 같이 **인자 값에 배열 포인터를 넣어 주어야 합니다.**

6. (추가조사) 구조체 -> , . 연산자 활용 예제

. 연산자 활용

```c
#include <stdio.h>
#include <string.h>    // strcpy 함수가 선언된 헤더 파일

struct Person {   // 구조체 정의
    char name[20];        // 구조체 멤버 1
    int age;              // 구조체 멤버 2
    char address[100];    // 구조체 멤버 3
};

int main()
{
    struct Person p1;     // 구조체 변수 선언

    // 점으로 구조체 멤버에 접근하여 값 할당
    strcpy(p1.name, "박태인");
    p1.age = 33;
    strcpy(p1.address, "수원시 권선구 평동");

    // 점으로 구조체 멤버에 접근하여 값 출력
    printf("이름: %s\n", p1.name);       // 이름: 박태인
    printf("나이: %d\n", p1.age);        // 나이: 33
    printf("주소: %s\n", p1.address);    // 주소: 수원시 권선구 평동

    return 0;
}
```

```
이름: 박태인
나이: 33
주소: 수원시 권선구 평동
```

6. (추가조사) 구조체 -> , . 연산자 활용 예제

->연산자 활용, 구조체 포인터 선언하고 메모리 할당

```c
#include <stdio.h>
#include <string.h>    // strcpy 함수가 선언된 헤더 파일
#include <stdlib.h>    // malloc, free 함수가 선언된 헤더 파일

struct Person {    // 구조체 정의
    char name[20];        // 구조체 멤버 1
    int age;              // 구조체 멤버 2
    char address[100];    // 구조체 멤버 3
};

int main()
{
    struct Person *p1 = malloc(sizeof(struct Person));    // 구조체 포인터 선언, 메모리 할당

    // 화살표 연산자로 구조체 멤버에 접근하여 값 할당
    strcpy(p1->name, "박태인");
    p1->age = 33;
    strcpy(p1->address, "수원시 권선구 평동");

    // 화살표 연산자로 구조체 멤버에 접근하여 값 출력
    printf("이름: %s\n", p1->name);       // 박태인
    printf("나이: %d\n", p1->age);        // 33
    printf("주소: %s\n", p1->address);    // 수원시 권선구 평동

    free(p1);    // 동적 메모리 해제

    return 0;
}
```

```
이름: 박태인
나이: 33
주소: 수원시 권선구 평동
```