Chapter 14. 포인터와 함수에 대한 이해

Chapter 14-1. 함수 인자로 배열 전달하기

인자전달의 기본방식은 값의 복사이다!

```
int SimpleFunc(int num) { . . . . }
int main(void)
{
    int age=17;
    SimpleFunc(age);
    . . . .
}
```

age에 저장된 값이 매개변수 num에 복사가 된다.

실제 전달되는 것은 age가 아니라 age에 **저장된 값**이다.

배열을 함수의 매개변수에 전달하는 이유는 함수 내에서 배열에 저장된 값을 참조하도록 하기 위함이다. 그런데 배열을 통째로 전달하지 않아도 이러한 일이 가능하다.

위의 코드에서 보이는 바와 같이, 배열을 함수의 인자로 전달하려면 배열을 통째로 복사할 수 있도록 배열이 매개변수로 선언되어야 한다. 그러나 **C언어는 매개변수로 배열의 선언을 허용하지 않는다.** 결론! 배열을 통째로 복사하는 방법은 C언어에 존재하지 않는다.

따라서 배열을 통째로 복사해서 전달하는 방식 대신에, **배열의 주소 값**을 전달하는 방식을 대신 취한다.

배열을 함수의 인자로 전달하는 방식

```
int main(void)
{
    int arr[3]={1, 2, 3};
    int * ptr=arr;
    . . . .
}
```

배열의 이름은 int형 포인터!

배열의 이름은 **int형 포인터!** 따라서 **int형 포인터** 변수에 배열의 이름이 지니는 주소 값을 저장할 수 있다.

위의 예제를 통해서 다음과 같은 코드의 구성이 가능함을 유추할 수 있다.

```
int main(void)
{
    int arr[3]={1, 2, 3};
    SimpleFunc(arr);
    . . . .
}
```

배열이름 arr이 지니는 주소 값의 전달

```
void SimpleFunc(int * param)
{
    printf("%d %d", param[0], param[1]);
}
```

배열 이름 arr은 **int형 포인터**이므로 매개변수는 **int형 포인터 변수**!

포인터 변수를 이용해서도 배열의 형태로 접근가능!

배열을 함수의 인자로 전달하는 예제

```
void ShowArayElem(int * param, int len)
{
    int i;
    for(i=0; i<len; i++)
        printf("%d ", param[i]);
    printf("\n");
}

int main(void)
{
    int arr1[3]={1, 2, 3};
    int arr2[5]={4, 5, 6, 7, 8};
    ShowArayElem(arr1, sizeof(arr1) / sizeof(int));
    ShowArayElem(arr2, sizeof(arr2) / sizeof(int));
    return 0;
}
```

실행결과

```
1 2 3
4 5 6 7 8
```

ArrayParam.c

ArrayParamAccess.c

```
void ShowArayElem(int * param, int len)
{
    int i;
    for(i=0; i<len; i++)
        printf("%d ", param[i]);
    printf("\n");
}
void AddArayElem(int * param, int len, int add)
{
    int i;
    for(i=0; i<len; i++)
        param[i] += add;
}

int main(void)
{
    int arr[3]={1, 2, 3};
    AddArayElem(arr, sizeof(arr) / sizeof(int), 1);
    ShowArayElem(arr, sizeof(arr) / sizeof(int));
    AddArayElem(arr, sizeof(arr) / sizeof(int), 2);
    ShowArayElem(arr, sizeof(arr) / sizeof(int));
    AddArayElem(arr, sizeof(arr) / sizeof(int), 3);
    ShowArayElem(arr, sizeof(arr) / sizeof(int));
    return 0;
}
```

실행결과

```
2 3 4
4 5 6
7 8 9
```

배열을 함수의 인자로 전달받는 함수의 또 다른 선언

```
void ShowArayElem (int * param, int len) { . . . . }
void AddArayElem (int * param, int len, int add) { . . . . }
```

동일한 선언

```
void ShowArayElem (int param[], int len) { . . . . }
void AddArayElem (int param[], int len, int add) { . . . . }
```

"매개변수의 선언"에서는 int * param과 int param[]이 동일한 선언이다. 따라서 배열을 인자로 전달 받는 경우에는 int param[]이 더 의미있어 보이므로 주로 사용된다.

```
int main(void)
{
    int arr[3]={1, 2, 3};
    int * ptr=arr;    // int ptr[]=arr; 로 대체 불가능
    . . . .
}
```

하지만 그 이외의 영역에서는 int * ptr의 선언을 int ptr[]으로 대체할 수 없다.

Chapter 14-2. Call-by-value vs. Call-by-reference

값을 전달하는 형태의 함수호출: Call-by-value

함수를 호출할 때 단순히 값을 전달하는 형태의 함수호출을 가리켜 Call-by-value라 하고, 메모리의 접근에 사용되는 주소 값을 전달하는 형태의 함수호출을 가리켜 Call-by-reference라 한다. 즉, Call-by-value와 Call-by-reference를 구분하는 기준은 함수의 인자로 전달되는 대상에 있다.

```
void NoReturnType(int num)
{
    if(num<0)
        return;
    . . . .
}
```

call-by-value

```
void ShowArayElem(int * param, int len)
{
    int i;
    for(i=0; i<len; i++)
        printf("%d ", param[i]);
    printf("\n");
}
```

call-by-reference

call-by-value와 call-by-reference라는 용어를 기준으로 구분하는 것이 중요한 게 아니다.

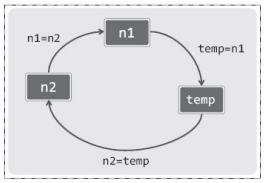
중요한 것은 각 함수의 특징을 이해하고 적절한 형태의 함수를 정의하는 것이다.

call-by-value 형태의 함수에서는 함수 외부에 선언된 변수에 접근이 불가능하다. 그러나 call-by-reference 형태의 함수에서는 외부에 선언된 변수에 접근이 가능하다.

잘못 적용된 Call-by-value

CallByValueSwap.c

```
void Swap(int n1, int n2)
{
    int temp=n1;
    n1=n2;
    n2=temp;
    printf("n1 n2: %d %d \n", n1, n2);
}

int main(void)
{
    int num1=10;
    int num2=20;
    printf("num1 num2: %d %d \n", num1, num2);

    Swap(num1, num2);   // num1과 num2에 저장된 값이 서로 바뀌길 기대!
    printf("num1 num2: %d %d \n", num1, num2);
    return 0;
}
```

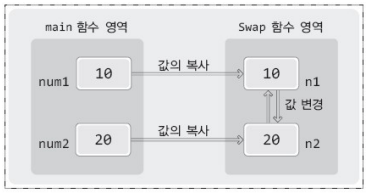


Swap 함수 내에서의 값의 교환

```
num1 num2: 10 20
n1 n2: 20 10
num1 num2: 10 20
```

call-by-value가 적절치 않은 경우

실행결과

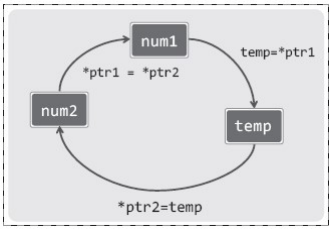


Swap 함수 내에서의 값의 교환은 외부에 영향을 주지 않는다.

주소 값을 전달하는 형태의 함수호출: Call-by-reference

CallByReferenceSwap.c

```
void Swap(int * ptr1, int * ptr2)
{
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

int main(void)
{
    int num1=10;
    int num2=20;
    printf("num1 num2: %d %d \n", num1, num2);
    Swap(&num1, &num2);
    printf("num1 num2: %d %d \n", num1, num2);
    return 0;
}
```



Swap 함수 내에서 함수 외부에 있는 변수간 값의 교환

```
num1 num2: 10 20
num1 num2: 20 10
```

실행결과

Swap 함수 내에서의 *ptr1은 main 함수의 num1
Swap 함수 내에서의 *ptr2는 main 함수의 num2
를 의미하게 된다.

scanf 함수호출 시 & 연산자를 붙이는 이유는?

```
int main(void)
{
    int num;
    scanf("%d", &num);
    . . . .
}
```

변수 num 앞에 & 연산자를 붙이는 이유는?

scanf 함수 내에서 외부에 선언된 변수 num에 접근 하기 위해서는 num의 주소 값을 알아야 한다. 그래서 scanf 함수는 변수의 주소 값을 요구한다.

```
int main(void)
{
    char str[30];
    scanf("%s", str);
    . . . .
}
```

배열 이름 str 앞에 & 연산자를 붙이지 않는 이유는?

str은 배열의 이름이고 그 자체가 주소 값이기 때문에 & 연산자를 붙이지 않는다. str을 전달함은 scanf 함수 내부로 배열 str의 주소 값을 전달하는 것이다.

13장 실습문제 (Lab 4)

- 변수 형태의 문자열의 이름을 str1으로 정의하고 키보드로부터 문자열을 입력 받은 후 해당 문자열이 제대로 입력되었는지 출력해 본다.
- 상수 형태의 문자열의 이름을 str2로 정의하고 program 상에서 문자열 하나를 입력 (예, Communications)하고 해당 문자열이 제대로 들어있는지 출력해 본다.
- str2 문자열을 str1에 복사하고 제대로 복사되었는지 출력해본다.

```
char str1[50];
char *str2 = "Communications";

// 복사 원리 ➔  str1[i] = str2[i];
```

```
문자열을 입력하세요 : Information
입력한 문자열 str1은 Information입니다.
str2 문자열에는 Communications가 있습니다.
str2를 str1에 복사한 후 str1 문자열을 출력하면 Communications입니다.
```

실습 문제 (Lab 1)

- 변수 num에 저장된 값의 제곱을 계산하는 함수를 구현하라. (두 가지 형태로 구현할 것)
  - **(1) Call-by-value 기반**의 SquareByValue 함수
    - int SquareByValue(int a);
  - **(2) Call-by-reference 기반**의 SquareByReference 함수
    - void SquareByReference(int *a);

```
<< main() 함수 내>>

int num = 20;
int num2 = 3000;

printf("Square value of %d is %d \n", num, SquareByValue(num));

SquareByReference(&num);
printf("Square value is %d \n", num);
```

13

실습 문제 (Lab 2)

- 세 변수에 저장된 값을 서로 뒤바꾸는 함수를 구현하라.
  - 다음 형태로 호출할 것: Swap3 (&a, &b, &c);
  - 치환 방법: 변수 a에 저장된 값은 변수 b에, 변수 b에 저장된 값은 변수 c에, 그리고 변수 c에 저장된 값은 변수 a에 저장되어야 함
  - Call-by-reference 특성을 사용할 것.

14

실습 시간 (2019년 9월 24일)

- 예제1 (13장): TwoStringType.c, PointerArray.c, StringArray.c
- 예제2 (14장): ArrayParam.c, ArrayParamAccess.c, CallByValueSwap.c, CallByReferenceSwap.c

- Lab 문제: 13장 Lab4.c, 14장 Lab1.c, Lab2.c

Chapter 14-3.포인터 대상의 const 선언

포인터 변수의 참조대상에 대한 const 선언

```
int main(void)
{
    int num=20;
    const int * ptr=&num;
    *ptr=30;    // 컴파일 에러!
    num=40;     // 컴파일 성공!
    . . . .
}
```

왼편의 const 선언이 갖는 의미

포인터 변수 ptr을 이용해서 ptr이 가리키는 변수에 저장된 값을 변경하는 것을 허용하지 않겠습니다!

그러나 변수 num에 저장된 값 자체의 변경이 불가능한 것은 아니다.
다만 ptr을 통한 변경을 허용하지 않을뿐이다.

포인터 변수의 상수화

```
int main(void)
{
    int num1=20;
    int num2=30;
    int * const ptr=&num1;
    ptr=&num2;      // 컴파일 에러!
    *ptr=40;        // 컴파일 성공!
    . . . .
}
```

왼편의 const 선언이 갖는 의미

포인터 변수 ptr에 저장된 값을 상수화 하겠다. 즉, ptr에 저장된 값은 변경이 불가능하다. ptr이 가리키는 대상의 변경을 허용하지 않는다.

```
const int * ptr=&num;
int * const ptr=&num;
```

→

```
const int * const ptr=&num;
```

두 가지 const 선언을 동시에 할 수 있다.

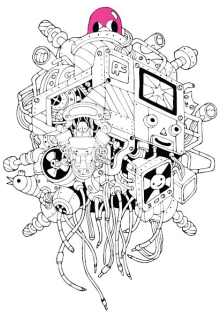
const 선언이 갖는 의미

```
int main(void)
{
     double PI=3.1415;
     double rad;
     PI=3.07;   // 실수로 잘못 삽입된 문장, 컴파일 시 발견 안됨
     scanf("%lf", &rad);
     printf("circle area %f \n", rad*rad*PI);
     return 0;
}
```

안전성이 높아진 코드

```
int main(void)
{
     const double PI=3.1415;
     double rad;
     PI=3.07;   // 컴파일 시 발견되는 오류상황
     scanf("%lf", &rad);
     printf("circle area %f \n", rad*rad*PI);
     return 0;
}
```

const 선언은 추가적인 기능을 제공하기 위한 것이 아니라, 코드의 안전성을 높이기 위한 것이다. 따라서 이러한 const의 선언을 소홀히하기 쉬운데, const의 선언과 같이 코드의 안전성을 높이는 선언은 가치가 매우 높은 선언이다.



Chapter 16. 다차원 배열

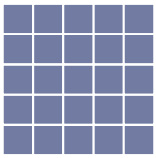
다차원 배열의 이해와 활용

2차원, 3차원 배열 OK! 4차원, 5차원 배열 NO!

```
int arrOneDim[10];          길이가 10인 1차원 int형 배열
int arrTwoDim[5][5];        가로, 세로의 길이가 각각 5인 2차원 int형 배열
int arrThreeDim[3][3][3];   가로, 세로, 높이의 길이가 각각 3인 3차원 int형 배열
```

1차원 배열 arrOneDim

3차원 배열
arrThreeDim



2차원 배열
arrTwoDim

문법적으로는 4차원 5차원 배열의 선언도 가능하지만 그것은 의미를 부여하기 힘든, 의미가 없는 배열이다.

다차원 배열을 의미하는 2차원 배열의 선언

2차원 배열의 선언 방식 → TYPE 배열이름[세로길이][가로길이];

int arr1[3][4];

int arr2[2][6];

TwoDimArrSize.c

```
int main(void)
{
    int arr1[3][4];
    int arr2[7][9];
    printf("세로3, 가로4: %d \n", sizeof(arr1));
    printf("세로7, 가로9: %d \n", sizeof(arr2));
    return 0;
}
```

실행결과

```
세로3, 가로4: 48
세로7, 가로9: 252
```

2차원 배열요소의 접근

```
int arr[3][3];
```

배열 생성


```
arr[0][0]=1;
```

0 0 접근


```
arr[0][1]=2;
```

0 1 접근

일반화

```
arr[N-1][M-1]=20;
printf("%d", arr[N-1][M-1]);
```

세로 N, 가로 M의 위치에 값을 저장 및 참조

```
arr[2][1]=5;
```

2 1 접근

2차원 배열요소 접근관련 예제

PopuResearch.c

```
int main(void)
{
    int villa[4][2];
    int popu, i, j;

    /* 가구별 거주인원 입력 받기 */
    for(i=0; i<4; i++)
    {
        for(j=0; j<2; j++)
        {
            printf("%d층 %d호 인구수: ", i+1, j+1);
            scanf("%d", &villa[i][j]);
        }
    }

    /* 빌라의 층별 인구수 출력하기 */
    for(i=0; i<4; i++)
    {
        popu=0;
        popu += villa[i][0];
        popu += villa[i][1];
        printf("%d층 인구수: %d \n", i+1, popu);
    }
    return 0;
}
```

실행결과

```
1층 1호 인구수: 2
1층 2호 인구수: 4
2층 1호 인구수: 3
2층 2호 인구수: 5
3층 1호 인구수: 2
3층 2호 인구수: 6
4층 1호 인구수: 4
4층 2호 인구수: 3
1층 인구수: 6
2층 인구수: 8
3층 인구수: 8
4층 인구수: 7
```

2차원 배열의 메모리상 할당의 형태

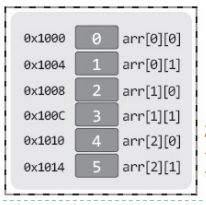
0x1001번지, 0x1002번지, 0x1003번지, 0x1004번지, 0x1005번지 . . . .

1차원적 메모리의 주소 값

0x12-0x24번지, 0x12-0x25번지, 0x12-0x26번지, 0x12-0x27번지 . . . .
0x13-0x24번지, 0x13-0x25번지, 0x13-0x26번지, 0x13-0x27번지 . . . .
0x14-0x24번지, 0x14-0x25번지, 0x14-0x26번지, 0x14-0x27번지 . . . .
. . . .

2차원적 메모리의 주소 값

실제 메모리는 1차원의 형태로 주소 값이 지정이 된다.
따라서 아래와 같은 형태로 2차원 배열의 주소 값이 지정된다.



2차원 배열의
실제 메모리
할당형태

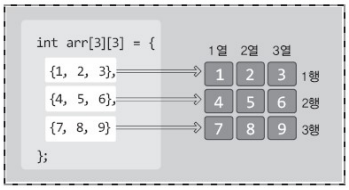
TwoDimArrAddr.c

```
int main(void)
{
    int arr[3][2];
    int i, j;
    for(i=0; i<3; i++)
        for(j=0; j<2; j++)
            printf("%p \n", &arr[i][j]);
    return 0;
}
```

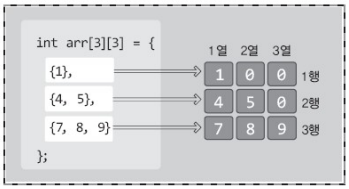
실행결과

```
002AFD54
002AFD58
002AFD5C
002AFD60
002AFD64
002AFD68
```

2차원 배열 선언과 동시에 초기화 하기



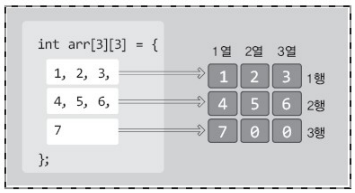
초기화 리스트 안에는 행 단위로 초기화할 값들을
별도의 중괄호로 명시한다.




```
int arr[3][3] = {
    {1, 0, 0},
    {4, 5, 0},
    {7, 8, 9}
};
```

채워지지 않은 빈 공간은 0으로 채워진다.

2차원 배열 선언과 동시에 초기화 하기 (계속)



별도의 중괄호를 사용하지 않으면 좌 상단부터 시작해서
우 하단으로 순서대로 초기화된다.

한 줄에 표현해도 된다.

```
int arr[3][3]={1, 2, 3, 4, 5, 6, 7};
```

마찬가지로 빈 공간은 0으로 채워진다.

```
int arr[3][3]={1, 2, 3, 4, 5, 6, 7, 0, 0};
```

2차원 배열 선언과 동시에 초기화 하기(예제)

TwoDimArrInit.c

```
int main(void)
{
    int i, j;

    /* 2차원 배열 초기화의 예 1 */
    int arr1[3][3]={
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    /* 2차원 배열 초기화의 예 2 */
    int arr2[3][3]={
        {1},
        {4, 5},
        {7, 8, 9}
    };
    /* 2차원 배열 초기화의 예 3 */
    int arr3[3][3]={1, 2, 3, 4, 5, 6, 7};

    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
            printf("%d ", arr1[i][j]);
        printf("\n");
    }
    printf("\n");

    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
            printf("%d ", arr2[i][j]);
        printf("\n");
    }
    printf("\n");

    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
            printf("%d ", arr3[i][j]);
        printf("\n");
    }
    return 0;
}
```

실행결과

```
1 2 3
4 5 6
7 8 9

1 0 0
4 5 0
7 8 9

1 2 3
4 5 6
7 0 0
```

배열의 크기를 알려주지 않고 초기화하기

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

두 개가 모두 비면 컴파일러가 채워 넣을 숫자를 결정하지 못한다.

8행 1열 ??
4행 2열 ??
2행 4열 ??

```
int arr1[][4]={1, 2, 3, 4, 5, 6, 7, 8};
int arr2[][2]={1, 2, 3, 4, 5, 6, 7, 8};
```

세로 길이만 생략할 수 있도록 약속

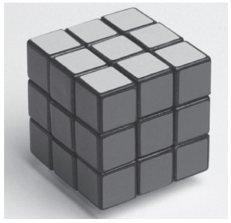
컴파일러가 세로 길이를 계산해 준다.

```
int arr1[2][4]={1, 2, 3, 4, 5, 6, 7, 8};
int arr2[4][2]={1, 2, 3, 4, 5, 6, 7, 8};
```

질문 있으신지요?

3차원 배열

3차원 배열의 논리적 구조



ThreeDimArraySize.c

```
int main(void)
{
    int arr1[2][3][4];
    double arr2[5][5][5];
    printf("높이2, 세로3, 가로4 int형 배열: %d \n", sizeof(arr1));
    printf("높이5, 세로5, 가로5 double형 배열: %d \n", sizeof(arr2));
    return 0;
}
```

실행결과

```
높이2, 세로3, 가로4 int형 배열: 96
높이5, 세로5, 가로5 double형 배열: 1000
```

int arr1[2][3][4];

높이 2, 세로 3, 가로 4인 int형 3차원 배열(세로 3, 가로 4인 배열이 두 개 겹친 형태)
2 x 3 x 4 = 24개의 room, 각 room의 크기는 4 bytes, 24 x 4 = 96 bytes

double arr2[5][5][5];

높이, 세로, 가로가 모두 5인 double형 3차원 배열(세로 5, 가로 5인 배열이 5개 겹친 형태)
5 x 5 x 5 = 125개의 room, 각 room의 크기는 8 bytes, 125 x 8 = 1,000 bytes

3차원 배열의 선언과 접근

ThreeDimArrayAccess.c

```
int main(void)
{
    int mean=0, i, j;
    int record[3][3][2]={
        {
            {70, 80},   // A 학급 학생 1의 성적
            {94, 90},   // A 학급 학생 2의 성적
            {70, 85}    // A 학급 학생 3의 성적
        },
        {
            {83, 90},   // B 학급 학생 1의 성적
            {95, 60},   // B 학급 학생 2의 성적
            {90, 82}    // B 학급 학생 3의 성적
        },
        {
            {98, 89},   // C 학급 학생 1의 성적
            {99, 94},   // C 학급 학생 2의 성적
            {91, 87}    // C 학급 학생 3의 성적
        }
    };
    for(i=0; i<3; i++)
        for(j=0; j<2; j++)
            mean += record[0][i][j];
    printf("A 학급 전체 평균: %g \n", (double)mean/6);

    mean=0;
    for(i=0; i<3; i++)
        for(j=0; j<2; j++)
            mean += record[1][i][j];
    printf("B 학급 전체 평균: %g \n", (double)mean/6);

    mean=0;
    for(i=0; i<3; i++)
        for(j=0; j<2; j++)
            mean += record[2][i][j];
    printf("C 학급 전체 평균: %g \n", (double)mean/6);
    return 0;
}
```

실행결과

```
A 학급 전체 평균: 81.5
B 학급 전체 평균: 83.3333
C 학급 전체 평균: 93
```

실습 문제 (Lab 1)

- 가로의 길이가 9, 세로의 길이가 3인 int형 2차원 배열을 선언하여 구구단 중 3단, 6단, 9단을 다음과 같이 저장하는 프로그램을 구현하라.
  - 2차원 모든 배열 값 저장하고, 확인을 위해 출력할 때 **2중 for 문**을 사용하라.

```
int gugudan[3][9];

for ( )
   for ( )
      gugudan[i][j] =  ;
```

	1열	2열	3열	4열	5열	6열	7열	8열	9열
1행	**3**	**6**	**9**	**12**	**15**	**18**	**21**	**24**	**27**
2행	**6**	**12**	**18**	**24**	**30**	**36**	**42**	**48**	**54**
3행	**9**	**18**	**27**	**36**	**45**	**54**	**63**	**72**	**81**

35

실습 문제 (Lab 2)

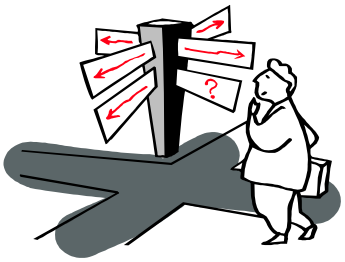
- **숙박관리 프로그램 작성 (Version 1)**
  - **char형 2차원 배열을 선언. (ex. room[층번호][방번호])**
  - **메뉴: 1.입실, 2.퇴실, 3.조회, 4.종료:**
  - **입실: 층 번호, 방 번호 입력, 고객 이름은 한 글자**
  - **퇴실: 층 번호, 방 번호 입력**
  - **2차원 배열은 '0'으로 초기화: '0'은 방이 비어있다는 의미.**
  - **모든 숫자 입력은 do~while()을 사용하여 경계 값 검사**

```
**********************
** 숙박관리 프로그램 **
**********************
1.입실, 2.퇴실, 3.조회, 4.종료:  1
층 번호 입력(1~3층): 1
방 번호 입력(1~5호): 1
고객이름(1자): A


1.입실, 2.퇴실, 3.조회, 4.종료:  3
A       0       0       0       0
0       0       0       0       0
0       0       0       0       0


1.입실, 2.퇴실, 3.조회, 4.종료:
```

실습 시간 (2019년 10월 1일)

- 예제1 (2차원): TwoDimArraySize.c, PopuResearch.c, TwoDimArrayAddr.c, TwoDimArrayInit.c,
- 예제2 (3차원): ThreeDimArraySize.c, ThreeDimArrayAccess.c

- Lab 문제: 16장 Lab1.c, Lab2.c



질문 있으신지요?