
포인터와 배열의 관계

- Chapter 08 -

학습목차

I. 포인터 디버깅

II. 포인터 응용

III. 메모리 사용

IV. 배열 응용

V. 포인터와 배열

VI. 포인터를 이용한 동적 배열

VII. goto문

포인터 디버깅

▶ 주소값 확인

▶ 디버거에서 포인터 확인

▷ 비주얼 스튜디오의 디버깅 기능을 사용하면 변수의 메모리 주소, 포인터, 역참조(*:포인터 연산자)를 확인 가능

```
#include<stdio.h>

int main(void) {

    int num = 10;
    int* numPtr = &num;

    *numPtr = 20;    //break point

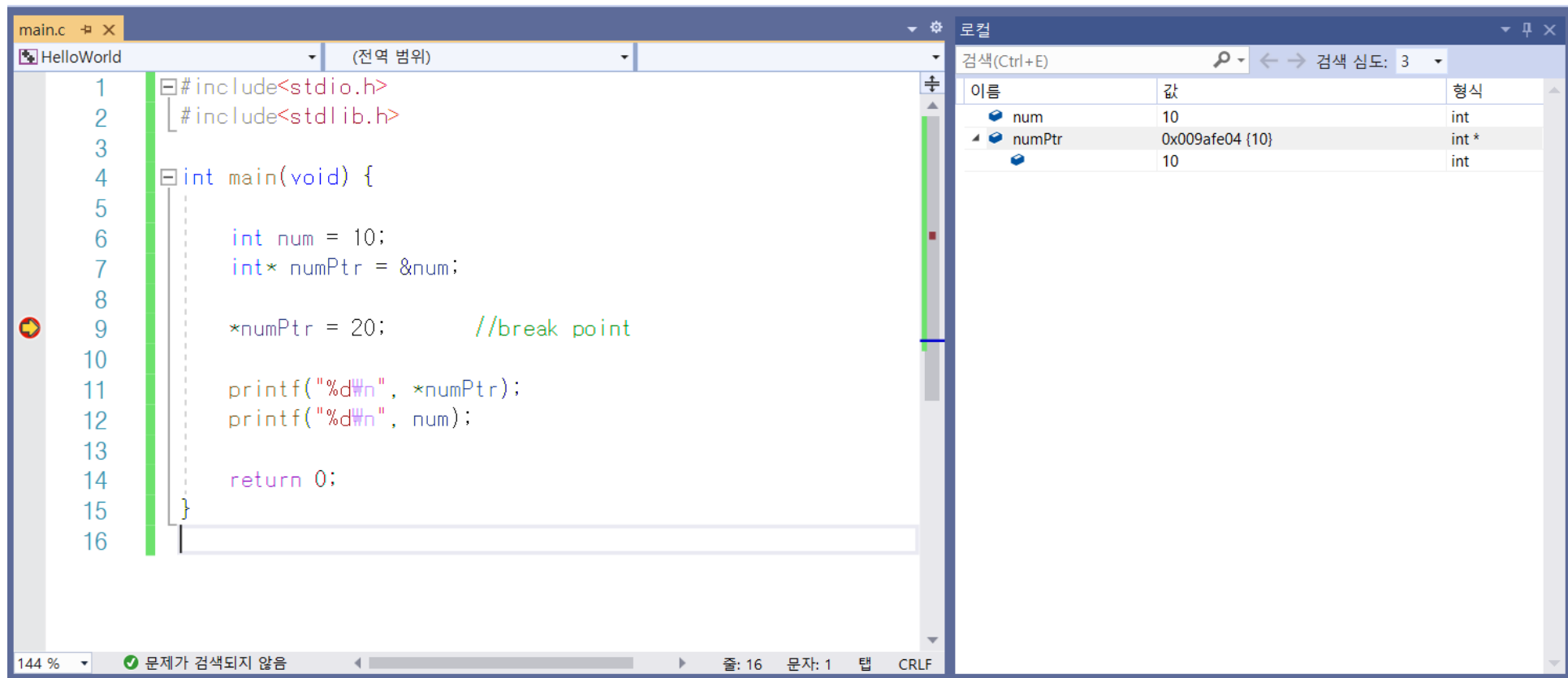
    printf("%d\n", *numPtr);
    printf("%d\n", num);

    return 0;
}
```

포인터 디버깅

▶ 디버거에서 포인터 확인하기

- ▶ *numPtr = 20;이 있는 줄에서 F9 키를 눌러 중단점을 설정하고 F5 키를 눌러 디버깅을 시작 후 화면 아래쪽에 지역 탭을 클릭
- ▶ 현재 코드는 numPtr = &num까지 실행된 상황



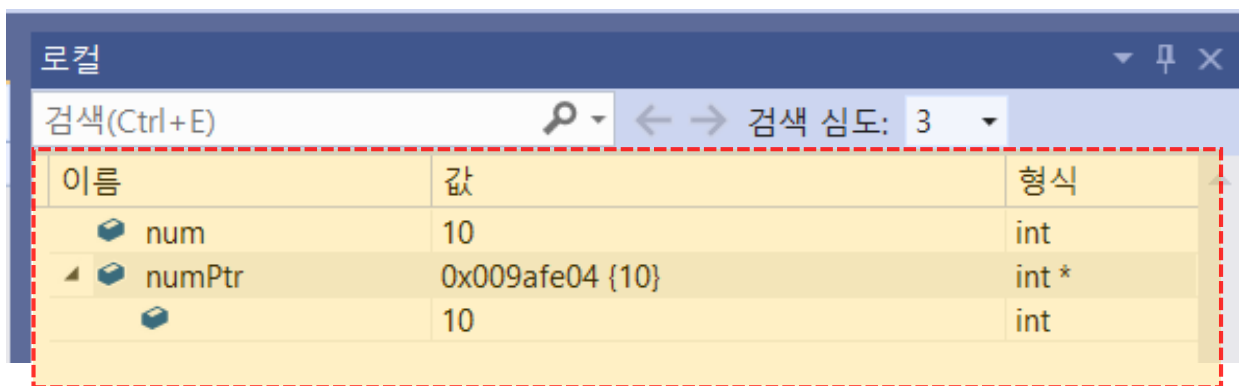
포인터 디버깅

▶ 디버거에서 포인터 확인하기

▶ 포인터 변수에 주소 저장 확인

▶ 메모리 주소는 프로그램을 실행할 때마다 달라짐

▶ 포인터 변수에서 ▶ 아이콘을 클릭하면 포인터가 가리키고 있는 변수에 저장된 값을 확인할 수 있음

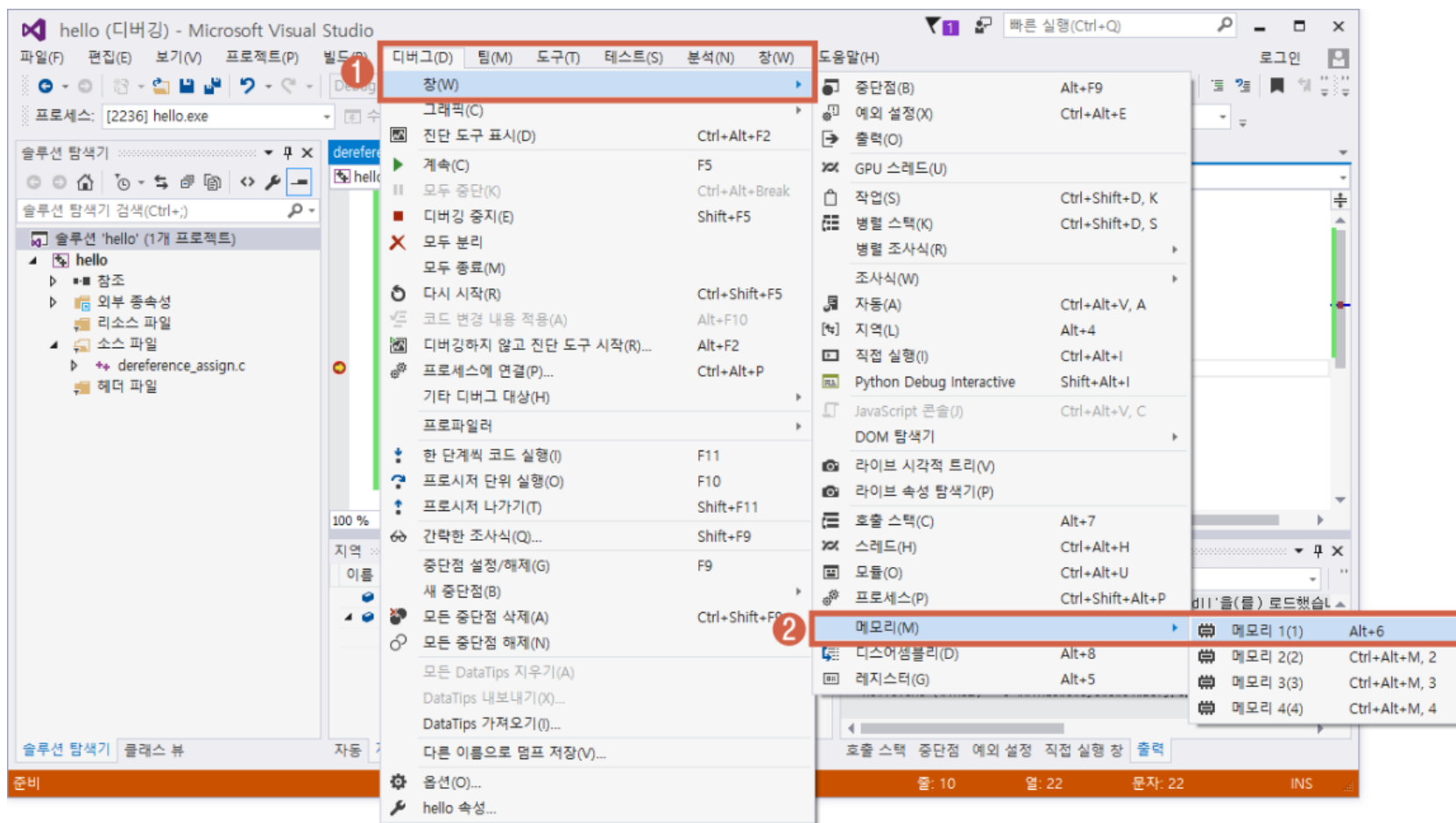


이름	값	형식
num	10	int
numPtr	0x009afe04 {10}	int *
	10	int

포인터 디버깅

▶ 메모리 내용을 직접 확인

▶ 메뉴에서 메모리 탭 열기 : 디버그 - 창 - 메모리 - 메모리(1)



포인터 디버깅

▶ 디버거에서 포인터 확인하기

- ▶ 포인터에 저장된 메모리 주소 복사 후 붙여넣기
- ▶ 메모리에 저장된 데이터 확인

로컬		
검색(Ctrl+E) 🔍 ⏪ ⏩ 검색 심도: 3		
이름	값	형식
num	10	int
numPtr	0x009afe04	int *
	10	int

메모리 1		
주소: 0x009AFE04 ↕ ↻ 열: 자동		
0x009AFE04	0a 00 00 00	cc cc cc cc 2c fe 9a 00 83 20 81 00 01 00 00 00 51 e7 00 48 56 e7 00 01 00 00 00???,???.? ?.....Q?.HV?.....
0x009AFE24	00 51 e7 00 48 56 e7 00 88 fe 9a 00 d7 1e 81 00 0c 1b 66 09 39 13 81 00 39 13 81 00 00 b0 b8 00	.Q?.HV?.???.?..f.9.?..9.?.??.
0x009AFE44	00 00
0x009AFE64	94 a5 81 00 a0 a5 81 00 00 00 00 00 34 fe 9a 00 00 00 00 00 f4 fe 9a 00 f0 38 81 00 54 6b 7d 09	???.???.4???.???.?8?.Tk}.
0x009AFE84	00 00 00 00 90 fe 9a 00 6d 1d 81 00 98 fe 9a 00 08 21 81 00 a8 fe 9a 00 59 63 ff 75 00 b0 b8 00	...???.m.?.???.!?.???.Yc.u.??.
0x009AFA04	40 63 ff 75 04 ff 9a 00 14 7c f0 76 00 b0 b8 00 0a c7 e3 90 00 00 00 00 00 00 00 00 00 b0 b8 00	@c.u..?..!?v.??..???.??.
0x009AFEC4	00 00
0x009AFEE4	00 00 00 00 00 00 00 00 b4 fe 9a 00 00 00 00 00 0c ff 9a 00 40 a0 f1 76 a6 5b 83 e6 00 00 00 00???.???.?@??v?{??...
0x009AFF04	14 ff 9a 00 e4 7b f0 76 ff ff ff ff fa 8f f2 76 00 00 00 00 00 00 00 00 00 39 13 81 00 00 b0 b8 00	..??.{?v....??v.....9.?.??.

포인터 디버깅

▶ 디버거에서 포인터 확인하기

▶ 코드 실행에 따른 데이터 변화 확인

The screenshot shows a C program in a debugger. The code is as follows:

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main(void) {
5
6     int num = 10;
7     int* numPtr = &num;
8
9     *numPtr = 20;    //break point
10
11     printf("%d\n", *numPtr); 경과 시간 1ms 이하
12     printf("%d\n", num);
13 }
```

The debugger is set to a break point at line 9. The local variable window on the right shows the following data:

이름	값	형식
num	20	int
numPtr	0x009afe04 {20}	int *
	20	int

The screenshot shows the memory window with the address 0x009AFE04. The memory contents are displayed in hexadecimal and ASCII. The first row shows the address 0x009AFE04 and the value 14, which corresponds to the character '2' in ASCII.

주소	0x009AFE04
0x009AFE04	14 00 00 00 cc cc cc cc 2c fe 9a 00 83 20 81 00 01 00 00 00 00 51 e7 00 48 56 e7 00 01 00 00 00
0x009AFE24	00 51 e7 00 48 56 e7 00 88 fe 9a 00 d7 1e 81 00 0c 1b 66 09 39 13 81 00 39 13 81 00 00 b0 b8 00
0x009AFE44	00 00
0x009AFE64	94 a5 81 00 a0 a5 81 00 00 00 00 00 34 fe 9a 00 00 00 00 00 f4 fe 9a 00 f0 38 81 00 54 6b 7d 09
0x009AFE84	00 00 00 00 90 fe 9a 00 6d 1d 81 00 98 fe 9a 00 08 21 81 00 a8 fe 9a 00 59 63 ff 75 00 b0 b8 00
0x009AFEa4	40 63 ff 75 04 ff 9a 00 14 7c f0 76 00 b0 b8 00 0a c7 e3 90 00 00 00 00 00 00 00 00 00 b0 b8 00

실습문제 01

▶ 포인터 복습 문제

- ▶ 두 개의 double형 변수 a, b에 다음과 같이 값이 저장되어 있다. 두 변수 a, b를 가리키는 포인터 변수를 사용하여 두 변수의 값을 바꾸는 프로그램을 작성하시오. swap함수를 작성하고 이를 이용하여 두 수를 변경하시오.

▷ double a = 1.3;

▷ double b = 1.7;

▶ 실행화면

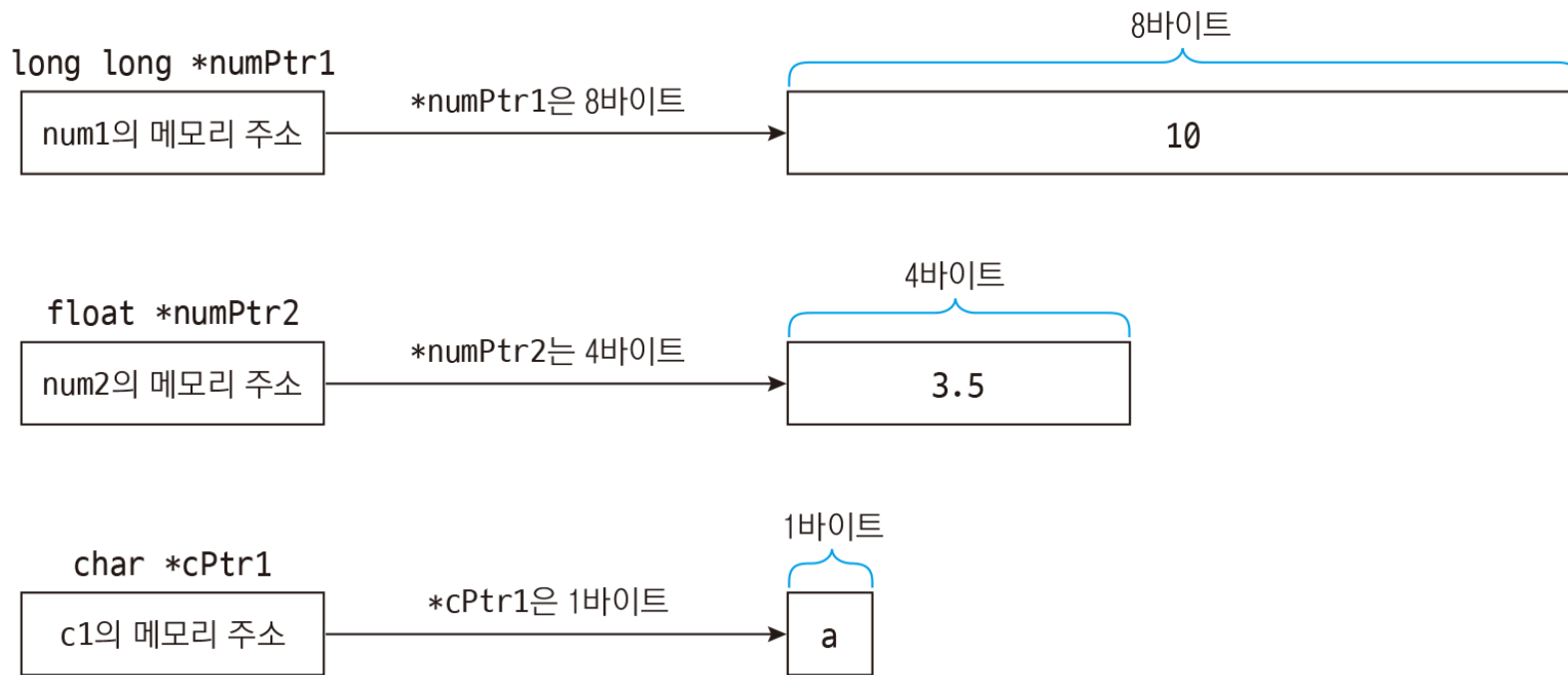
변경 전 - a : 1.3, b : 1.7

변경 후 - a : 1.7, b : 1.3

void 포인터

▶ 포인터

- ▶ 포인터 변수의 타입마다 접근 방식이 다르기 때문에 다양한 타입의 포인터를 사용
 - ▷ 포인터 연산자로 포인터 변수가 가리키고 있는 변수에 저장된 값에 접근할 경우 변수의 자료형의 크기에 따라 맞춰서 접근
 - ▷ int형 포인터는 4byte, char형 포인터는 1byte, double형 포인터는 8byte만큼 값을 가져오거나 저장
 - ▷ C언어에서는 문법상, 자료형이 다른 포인터끼리 메모리 주소를 저장하면 컴파일 오류가 발생



void 포인터

▶ void형 포인터

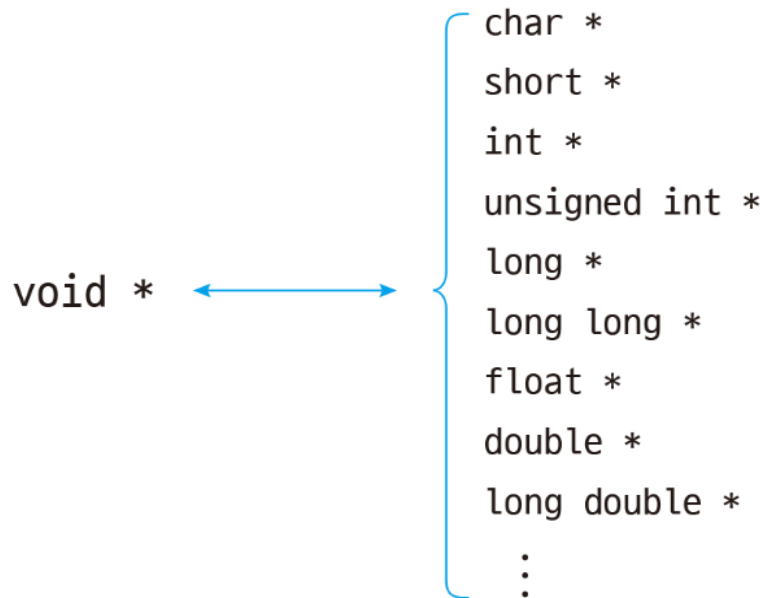
▶ void형 포인터는 자료형이 정해지지 않은 포인터

▷ void형 포인터는 자료형이 정해지지 않은 특성으로 인하여 어떤 자료형이든 저장 가능

▷ 반대로 어떠한 자료형이든 void 포인터를 저장 가능

▷ 이러한 특성으로 인하여 void형 포인터를 범용 포인터라고 부름

▶ void형 포인터는 직접 자료형을 변환하지 않아도 암시적으로 자료형이 변환됨



void 포인터

▶ void 포인터

▶ 포인터 변수의 타입마다 접근 방식이 다르기 때문에 다양한 타입의 포인터를 사용

▷ 포인터 연산자로 포인터 변수가 가리키고 있는 변수에 저장된 값에 접근할 경우 변수의 자료형의 크기에 따라 맞춰서 접근

▶ 포인터의 선언

▷ void *포인터이름;

```
int main(){
    int num1 = 10;
    char c1 = 'a';
    int *numPtr1 = &num1;
    char *cPtr1 = &c1;
    void *ptr;           // void 포인터 선언
    // 포인터 자료형이 달라도 컴파일 경고가 발생하지 않음
    ptr = numPtr1;       // void 포인터에 int 포인터 저장
    ptr = cPtr1;         // void 포인터에 char 포인터 저장
    // 포인터 자료형이 달라도 컴파일 경고가 발생하지 않음
    numPtr1 = ptr;       // int 포인터에 void 포인터 저장
    cPtr1 = ptr;         // char 포인터에 void 포인터 저장
    return 0;
}
```

void 포인터

▶ void형 포인터의 역참조

- ▶ void 포인터는 자료형이 정해지지 않았으므로 값을 가져오거나 저장할 크기도 정해지지 않음
- ▶ 또한 void형의 변수를 선언할 수도 없으므로 void 포인터는 역참조 불가

```
ptr = numPtr1;    // void 포인터에 int 포인터 저장
printf("%d", *ptr); // void 포인터는 역참조할 수 없음. 컴파일 에러

ptr = cPtr1;      // void 포인터에 char 포인터 저장
printf("%c", *ptr); // void 포인터는 역참조할 수 없음. 컴파일 에러
```

```
error C2100: 간접 참조가 잘못되었습니다.
error C2100: 간접 참조가 잘못되었습니다.
```

```
void v1;    // void로는 변수를 선언할 수 없음. 컴파일 에러
```

```
error C2182: 'v1': 'void' 형식을 잘못 사용했습니다.
```

void 포인터

- ▶ void형 포인터를 사용하는 이유
 - ▶ 함수에서 다양한 자료형을 받아들이는 경우
 - ▶ 함수의 반환 포인터를 다양한 자료형으로 된 포인터에 저장하는 경우
 - ▶ 처리하는 데이터의 자료형을 숨기고 싶을 경우 사용

메모리 사용

▶ 메모리 사용하기

- ▶ 지금까지 포인터에 변수의 메모리 주소를 저장해서 사용
- ▶ 포인터에 malloc 함수를 이용하여 원하는 만큼 메모리를 할당하여 사용 가능
 - ▶ 메모리의 할당 및 해제 함수는 stdlib.h 헤더 파일에 선언
- ▶ 메모리 사용 패턴



메모리 사용

▶ 메모리 할당하기

- ▶ malloc은 memory allocation을 줄여서 사용

- ▶ 포인터 = malloc(크기);

 - ▷ void *malloc(size_t _Size);

 - ▷ 성공하면 메모리 주소를 반환, 실패하면 NULL을 반환

- ▶ numPtr2 = malloc(sizeof(int));

 - ▷ 필요한 메모리 크기는 바이트 단위로 지정

- ▶ malloc 함수로 메모리 할당은 원하는 시점에 원하는 만큼 메모리를 할당할 수 있어서 동적 메모리 할당(dynamic memory allocation)이라 부름

메모리 사용

▶ 메모리 할당하기

▶ malloc으로 메모리를 할당한 후 해당 메모리 공간이 필요없어지면 반드시 free 함수로 메모리 해제를 해야함

▷ memory leak 발생

```
#include <stdio.h>
#include <stdlib.h>    // malloc, free 함수가 선언된 헤더 파일
int main(){
    int num1 = 20;    // int형 변수 선언
    int *numPtr1;      // int형 포인터 선언
    numPtr1 = &num1;  // num1의 메모리 주소를 구하여 numPtr에 할당
    int *numPtr2;      // int형 포인터 선언
    numPtr2 = malloc(sizeof(int));    // int의 크기 4바이트만큼 동적 메모리 할당
    printf("%p\n", numPtr1);    // 006BFA60: 변수 num1의 메모리 주소 출력. 컴퓨터마다, 실행할 때마다 달라짐
    printf("%p\n", numPtr2);    // 009659F0: 새로 할당된 메모리의 주소 출력. 컴퓨터마다, 실행할 때마다 달라짐
    free(numPtr2);    // 동적으로 할당한 메모리 해제
    return 0;
}
```

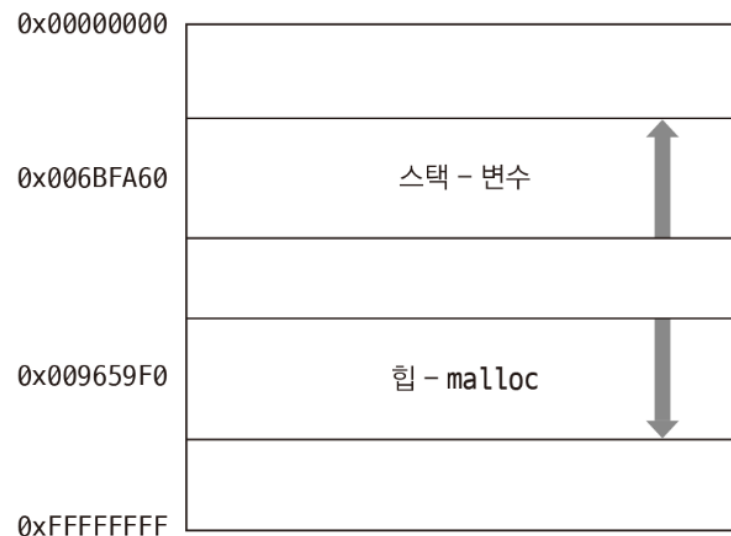
006BFA60	#메모리 주소. 컴퓨터마다, 실행할 때마다 달라짐
009659F0	#메모리 주소. 컴퓨터마다, 실행할 때마다 달라짐

메모리 사용

▶ 메모리 할당하기

▶ 스택(Stack)과 힙(heap)

- ▶ 일반 변수는 스택 영역에 생성되며 malloc으로 생성된 공간은 heap영역에 생성됨
- ▶ 스택에 생성된 변수는 메모리 해제 처리를 하지 않아도 되지만 힙에 메모리를 할당할 경우 반드시 해제해야 함



▶ free(포인터);

▶ void free(void *_Block);

▶ free(numPtr2);

메모리 사용

▶ 메모리에 값 저장

▶ 메모리에 데이터를 저장할 경우 *연산자를 사용

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

int main()
{
    int *numPtr;    // int형 포인터 선언

    numPtr = malloc(sizeof(int));    // int의 크기 4바이트만큼 동적 메모리 할당

    *numPtr = 10;    // 포인터를 역참조한 뒤 값 할당

    printf("%d\n", *numPtr);    // 10: 포인터를 역참조하여 메모리에 저장된 값 출력

    free(numPtr);    // 동적 메모리 해제

    return 0;
}
```

10

메모리 사용

▶ 메모리 내용을 한꺼번에 설정하기

▶ memset(메모리 시작 포인터, 채우고자 하는 값, 채우고자 하는 바이트 크기);

▷ void *memset(void *_Dst, int _Val, size_t _Size);

▷ 어떤 메모리의 시작점부터 연속된 범위를 어떤 값으로(바이트 단위) 모두 지정하고 싶을 때 사용하는 함수

▷ 값 설정이 끝난 포인터를 반환

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

int main()
{
    long long *numPtr = malloc(sizeof(long long)); // long long의 크기 8바이트만큼 동적 메모리 할당
    memset(numPtr, 0x27, 8);    // numPtr이 가리키는 메모리를 8바이트만큼 0x27로 설정
    printf("0x%11lx\n", *numPtr);    // 0x2727272727272727: 27이 8개 들어가 있음
    free(numPtr);    // 동적으로 할당한 메모리 해제
    return 0;
}
```

0x2727272727272727

실습예제 02

▶ 메모리 할당하기

▶ 다음 소스 코드를 완성하여 입력된 두 정수의 합이 출력되도록 작성하시오.

▷ 표준 입력으로 두 정수를 입력 받음

▷ 입력 값의 범위는 0~1073741824

▷ 정답에는 밑줄 친 부분에 들어갈 코드만 작성

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int num1;
    int num2;

    _____

    scanf("%d %d", &num1, &num2);
    *numPtr1 = num1;
    *numPtr2 = num2;
    printf("%d\n", *numPtr1 + *numPtr2);
    free(numPtr1);
    free(numPtr2);
    return 0;
}
```

10	20	#입력
30		#출력

실습문제 03

▶ 배열 복습문제

▶ 아래 코드를 완성하여 배열에 저장된 점수의 평균을 출력하시오.

```
#include <stdio.h>

int main()
{
    float scores[10] = { 67.2f, 84.3f, 97.0f, 87.1f, 71.9f, 63.0f, 90.1f, 88.0f, 79.7f, 95.3f };
    float sum = 0.0f;
    float average;

    for (int i = 0; i < sizeof(scores) / sizeof(float); i++)
    {
        ① _____
    }

    ② _____

    printf("%f\n", average);

    return 0;
}
```

82.360001

실습문제 04

▶ 배열 복습문제

▶ 다음 소스 코드를 완성하여 배열에 저장된 2진수를 10진수로 출력하시오.

```
#include <stdio.h>

int main()
{
    int decimal = 0;
    int binary[4] = { 1, 1, 0, 1 };    // 1101 순서대로 저장됨

    _____
    ...
    _____

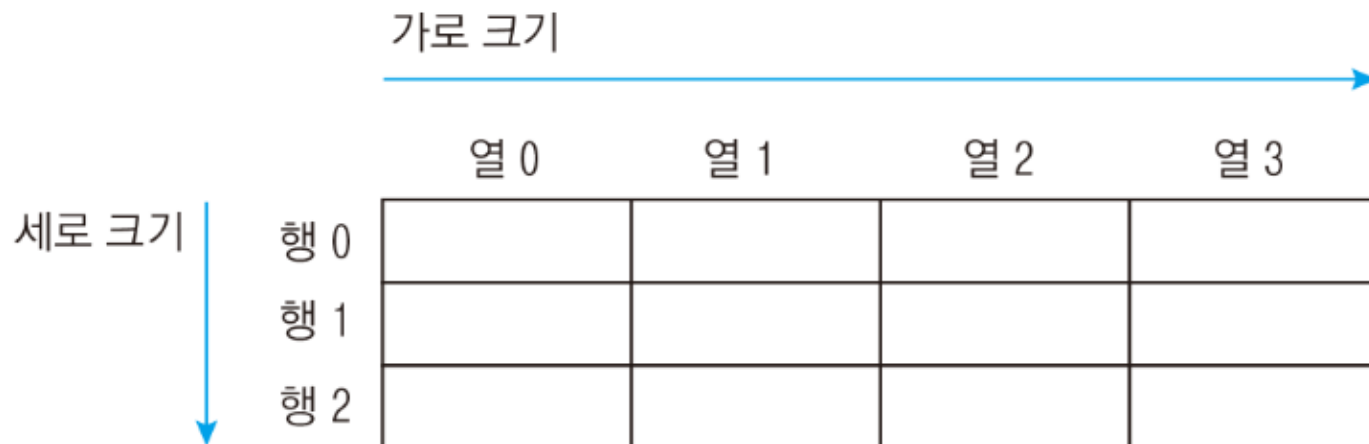
    printf("%d\n", decimal);

    return 0;
}
```

2차원 배열

▶ 2차원 배열

- ▶ 앞서 학습한 배열은 한 줄로 늘어난 1차원 배열
- ▶ 2차원 배열은 평면 구조
- ▶ 행과 열이 모두 0부터 시작
- ▶ 2차원 배열은 대괄호([])를 두 번 사용하여 선언
 - ▶ 자료형 배열이름[세로크기][가로크기];



The diagram illustrates a 2D array structure. It features a grid with 3 rows and 4 columns. Above the grid, a horizontal blue arrow points to the right, labeled '가로 크기' (Horizontal Size). To the left of the grid, a vertical blue arrow points downwards, labeled '세로 크기' (Vertical Size). The columns are labeled '열 0', '열 1', '열 2', and '열 3' from left to right. The rows are labeled '행 0', '행 1', and '행 2' from top to bottom. The grid itself is composed of 12 empty cells.

	열 0	열 1	열 2	열 3
행 0				
행 1				
행 2				

2차원 배열

▶ 2차원 배열을 선언하고 요소에 접근하기

▶ 2차원 배열의 초기화

▷ 자료형 배열이름[세로크기][가로크기] = {{ 값, 값, 값 }, { 값, 값, 값 } };

▷ 가로 요소들을 먼저 묶어주고 가로줄을 세로크기 만큼 다시 묶음

▷ 배열의 요소에 순서대로 저장되는 값은 세로, 가로 크기보다 적게 입력할 수 있지만 많으면 오류가 발생

```
int numArr[3][4] = {  
    { 가로 요소 4개 },  
    { 가로 요소 4개 },  
    { 가로 요소 4개 }  
}; //      ↑ 세로 3줄
```

▶ 2차원 배열의 접근

▷ 배열[세로 인덱스][가로 인덱스]

```
int num1 = numArr[1][2];    // 2차원 배열에서 세로 인덱스 1, 가로 인덱스 2인 요소에 접근
```

2차원 배열

▶ 2차원 배열을 선언하고 요소에 접근

```
#include <stdio.h>

int main()
{
    int numArr[3][4] = {    // 세로 크기 3, 가로 크기 4인 int형 2차원 배열 선언
        { 11, 22, 33, 44 },
        { 55, 66, 77, 88 },
        { 99, 110, 121, 132 }
    };

    // ↓ 세로 인덱스
    printf("%d\n", numArr[0][0]);    // 11 : 세로 인덱스 0, 가로 인덱스 0인 요소 출력
    printf("%d\n", numArr[1][2]);    // 77 : 세로 인덱스 1, 가로 인덱스 2인 요소 출력
    printf("%d\n", numArr[2][0]);    // 99 : 세로 인덱스 2, 가로 인덱스 0인 요소 출력
    printf("%d\n", numArr[2][3]);    // 132 : 세로 인덱스 2, 가로 인덱스 2인 요소 출력
    // ↑ 가로 인덱스

    return 0;
}
```

11
77
99
132

int numArr[3][4];

numArr[0][0] [0][1] [0][2] [0][3]			
↓	↓	↓	↓
11	22	33	44
55	66	77	88 ← numArr[1][3]
99	110	121	132
↑ numArr[2][0]		↑ numArr[2][3]	

2차원 배열

▶ 2차원 배열을 0으로 초기화

▶ 1차원 배열과 사용방법 동일

```
#include <stdio.h>

int main()
{
    int numArr[3][4] = { 0, };          // 2차원 배열의 요소를 모두 0으로 초기화

    printf("%d\n", numArr[0][0]);        // 0: 세로 인덱스 0, 가로 인덱스 0인 요소 출력
    printf("%d\n", numArr[1][2]);        // 0: 세로 인덱스 1, 가로 인덱스 2인 요소 출력
    printf("%d\n", numArr[2][0]);        // 0: 세로 인덱스 2, 가로 인덱스 0인 요소 출력
    printf("%d\n", numArr[2][3]);        // 0: 세로 인덱스 2, 가로 인덱스 3인 요소 출력

    return 0;
}
```

```
0
0
0
0
```

2차원 배열

▶ 2차원 배열의 요소에 값 할당

▶ 배열[세로인덱스][가로인덱스] = 값;

```
int main(){
    int numArr[2][3];
    numArr[0][0] = 1;    // 세로 인덱스 0, 가로 인덱스 0인 요소에 값 할당
    numArr[0][1] = 2;    // 세로 인덱스 0, 가로 인덱스 1인 요소에 값 할당
    numArr[0][2] = 3;    // 세로 인덱스 0, 가로 인덱스 2인 요소에 값 할당
    numArr[1][0] = 4;    // 세로 인덱스 1, 가로 인덱스 0인 요소에 값 할당
    numArr[1][1] = 5;    // 세로 인덱스 1, 가로 인덱스 1인 요소에 값 할당
    numArr[1][2] = 6;    // 세로 인덱스 1, 가로 인덱스 2인 요소에 값 할당
    printf("%d\n", numArr[0][0]);    // 1 : 세로 인덱스 0, 가로 인덱스 0인 요소 출력
    printf("%d\n", numArr[1][2]);    // 6 : 세로 인덱스 1, 가로 인덱스 2인 요소 출력
    return 0;
}
```

메모리 1

주소: 0x00D6FB6C

0x00D6FB6C	01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00 06 00 00 00 cc cc cc cc 0a 5a
0x00D6FB92	c6 00 01 00 00 00 00 51 12 01 48 56 12 01 01 00 00 00 51 12 01 48 56 12 01 08 fc d6 00
0x00D6FB88	39 13 c6 00 39 13 c6 00 00 30 eb 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x00D6FBDE	00 00 00 00 00 00 94 a5 c6 00 a0 a5 c6 00 00 00 00 00 b4 fb d6 00 00 00 00 74 fc d6 00 20 3e c6 00 56 2f 2a 10
0x00D6FC04	00 00 00 00 10 fc d6 00 7d 1e c6 00 18 fc d6 00 18 22 c6 00 28 fc d6 00 59 63 ff 75 00 30 eb 00 40 63 ff 75 84 fc
0x00D6FC2A	d6 00 14 7c f0 76 00 30 eb 00 b6 01 ea 07 00 00 00 00 00 00 00 00 00 30 eb 00 00 00 00 00 00 00 00 00 00
0x00D6FC50	00 34 fc d6 00 00 00 00 8c fc

로컬

검색(Ctrl+E)



검색 심도: 3

이름	값	형식
numArr	0x00d6fb6c {0x00d6fb6c {1, 2, 3}, 0x00d...	int[2][3]
[0]	0x00d6fb6c {1, 2, 3}	int[3]
[1]	0x00d6fb78 {4, 5, 6}	int[3]

2차원 배열

▶ 2차원 배열의 요소에 값 할당하기

▶ 2차원 배열에서 범위를 벗어난 인덱스에 접근할 경우

```
#include<stdio.h>
int main()
{
    int numArr[2][3] = { 0, };

    printf("%d\n", numArr[-1][-1]);    // 음수이므로 잘못된 인덱스
    printf("%d\n", numArr[0][4]);      // 가로 인덱스가 배열의 범위를 벗어남 - [1][0]에 접근
    printf("%d\n", numArr[4][0]);      // 세로 인덱스가 배열의 범위를 벗어남
    printf("%d\n", numArr[5][5]);      // 세로, 가로 인덱스 모두 배열의 범위를 벗어남

    return 0;
}
```

```
-858993460
0
8148552
11473721
```

2차원 배열

▶ 2차원 배열의 크기 구하기

```
#include <stdio.h>
int main(){
    int numArr[3][4] = {      // 세로 크기 3, 가로 크기 4인 int형 2차원 배열 선언
        { 11, 22, 33, 44 },
        { 55, 66, 77, 88 },
        { 99, 110, 121, 132 }
    };
    printf("%d\n", sizeof(numArr));    // 48: 4바이트 크기의 요소가 12(4*3)개이므로 48
    int col = sizeof(numArr[0]) / sizeof(int);    // 4: 2차원 배열의 가로 크기를 구할 때는
                                                    // 가로 한 줄의 크기를 요소의 크기로 나눠줌

    int row = sizeof(numArr) / sizeof(numArr[0]); // 3: 2차원 배열의 세로 크기를 구할 때는
                                                    // 배열이 차지하는 전체 공간을 가로 한 줄의 크기로 나눠줌

    printf("%d\n", col);    // 4
    printf("%d\n", row);    // 3
    return 0;
}
```

48

4

3

2차원 배열

▶ 반복문으로 2차원 배열의 요소를 모두 출력

```
int main(){
    int numArr[3][4] = {    // 세로 크기 3, 가로 크기 4인 int형 2차원 배열 선언
        { 11, 22, 33, 44 },
        { 55, 66, 77, 88 },
        { 99, 110, 121, 132 }
    };
    int col = sizeof(numArr[0]) / sizeof(int);    // 4: 2차원 배열의 가로 크기를 구할 때는
                                                    // 가로 한 줄의 크기를 요소의 크기로 나눠줌

    int row = sizeof(numArr) / sizeof(numArr[0]); // 3: 2차원 배열의 세로 크기를 구할 때는
                                                    // 배열이 차지하는 전체 공간을 가로 한 줄의 크기로 나눠줌
    for (int i = 0; i < row; i++)    // 2차원 배열의 세로 크기만큼 반복
    {
        for (int j = 0; j < col; j++)    // 2차원 배열의 가로 크기만큼 반복
        {
            printf("%d ", numArr[i][j]); // 2차원 배열의 인덱스에 반복문의 변수 i, j를 지정
        }
        printf("\n");    // 가로 요소를 출력한 뒤 다음 줄로 넘어감
    }
    return 0;
}
```

```
11 22 33 44
55 66 77 88
99 110 121 132
```

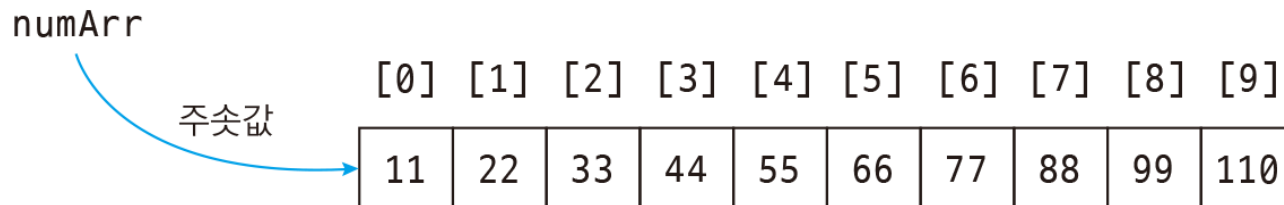
2차원 배열

▶ 배열을 포인터에 저장

▶ 배열의 이름은 포인터이며 첫번째 요소의 주소 값을 저장

```
int main(){
    int numArr[10] = { 11, 22, 33, 44, 55, 66, 77, 88, 99, 110 };    // 크기가 10인 int형 배열
    int *numPtr = numArr;      // 포인터에 int형 배열을 할당
    printf("%d\n", *numPtr);    // 11: 배열의 주소가 들어있는 포인터를 역참조하면 배열의
                                // 첫 번째 요소에 접근
    printf("%d\n", *numArr);    // 11: 배열 자체를 역참조해도 배열의 첫 번째 요소에 접근
    printf("%d\n", numPtr[5]);  // 66: 배열의 주소가 들어있는 포인터는 인덱스로 접근할 수 있음
    printf("%d\n", sizeof(numArr)); // 40: sizeof로 배열의 크기를 구하면 배열이 메모리에
                                // 차지하는 공간이 출력됨
    printf("%d\n", sizeof(numPtr)); // 4 : sizeof로 배열의 주소가 들어있는 포인터의 크기를
                                // 구하면 포인터의 크기가 출력됨(64비트라면 8)

    return 0;
}
```



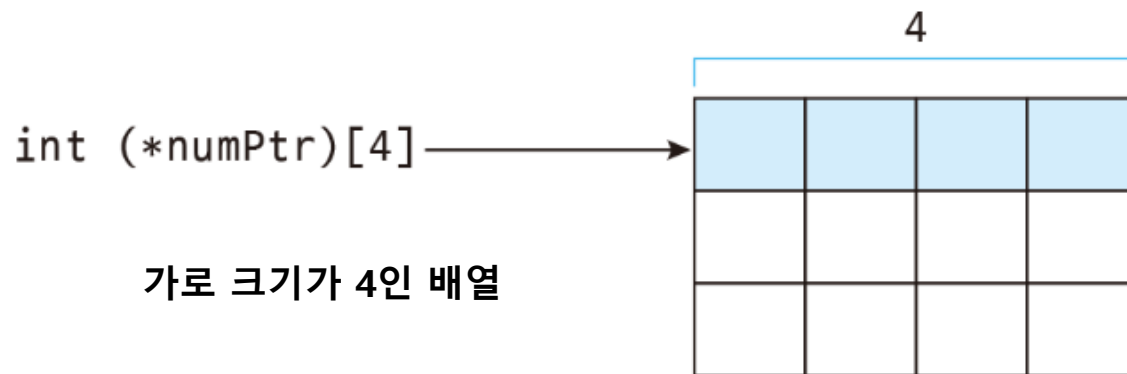
2차원 배열

▶ 2차원 배열을 포인터에 저장하는 방법

▶ 자료형 (*포인터이름)[가로크기];

▶ *과 포인터 이름을 괄호로 묶어준 뒤 대괄호에 2차원 배열의 가로 크기를 지정

```
int numArr[3][4] = { // 세로 크기 3, 가로 크기 4인 int형 2차원 배열 선언
    { 11, 22, 33, 44 },
    { 55, 66, 77, 88 },
    { 99, 110, 121, 132 }
};
```



2차원 배열

▶ 2차원 배열을 포인터에 저장하는 방법

```
int main(){
    int numArr[3][4] = {          // 세로 3, 가로 4 크기의 int형 2차원 배열 선언
        { 11, 22, 33, 44 },
        { 55, 66, 77, 88 },
        { 99, 110, 121, 132 }
    };
    int(*numPtr)[4] = numArr;

    printf("%p\n", *numPtr); // 002BFE5C: 2차원 배열 포인터를 역참조하면 세로 첫 번째의 주소가 나옴
                           // 컴퓨터마다, 실행할 때마다 달라짐
    printf("%p\n", &numArr[0][0]);
    printf("%d\n", numPtr[2][1]); // 110: 2차원 배열 포인터는 인덱스로 접근할 수 있음
    printf("%d\n", sizeof(numArr)); // 48: sizeof로 2차원 배열의 크기를 구하면 배열이 메모리에
                                   // 차지하는 공간이 출력됨
    printf("%d\n", sizeof(numPtr)); // 4 : sizeof로 2차원 배열 포인터의 크기를
                                   // 구하면 포인터의 크기가 출력됨(64비트라면 8)

    return 0;
}
```

```
00CFFA14
00CFFA14
110
48
4
```

3차원 배열

▶ 3차원 배열

▶ 3차원 배열은 높이 x 가로 x 세로 형태로 구성

▶ 자료형 배열이름[높이][세로크기][가로크기];

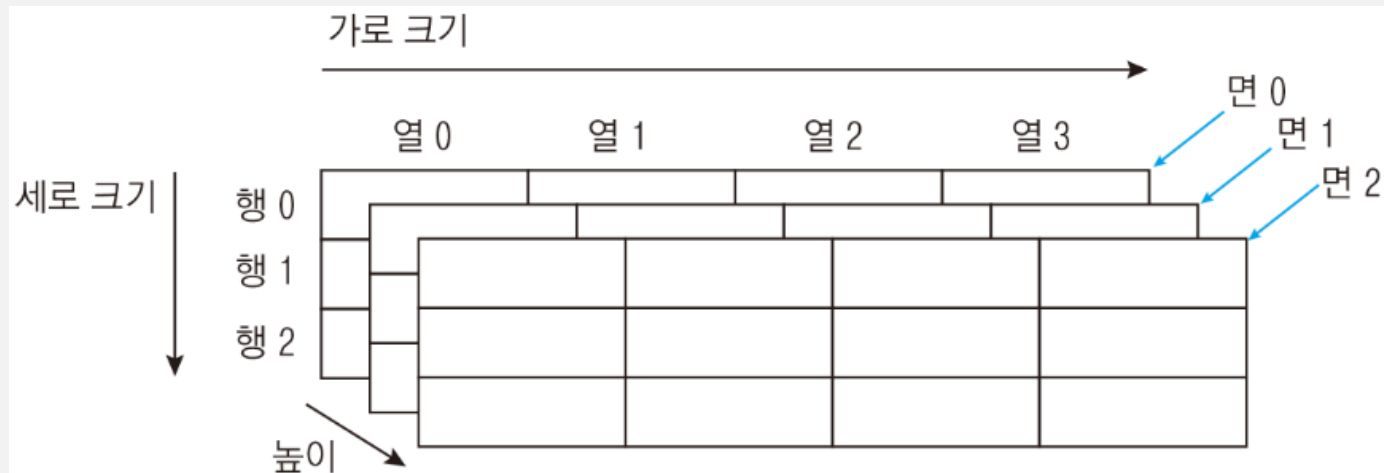
▶ 3차원 배열 접근

▷ 배열[높이인덱스][세로인덱스][가로인덱스];

▷ 배열[높이인덱스][세로인덱스][가로인덱스] = 값;

```
int numArr[2][3][4] = {  
    {  
        { 11, 22, 33, 44 },  
        { 55, 66, 77, 88 },  
        { 99, 110, 121, 132 }  
    },  
    {  
        { 111, 122, 133, 144 },  
        { 155, 166, 177, 188 },  
        { 199, 1110, 1121, 1132 }  
    }  
};
```

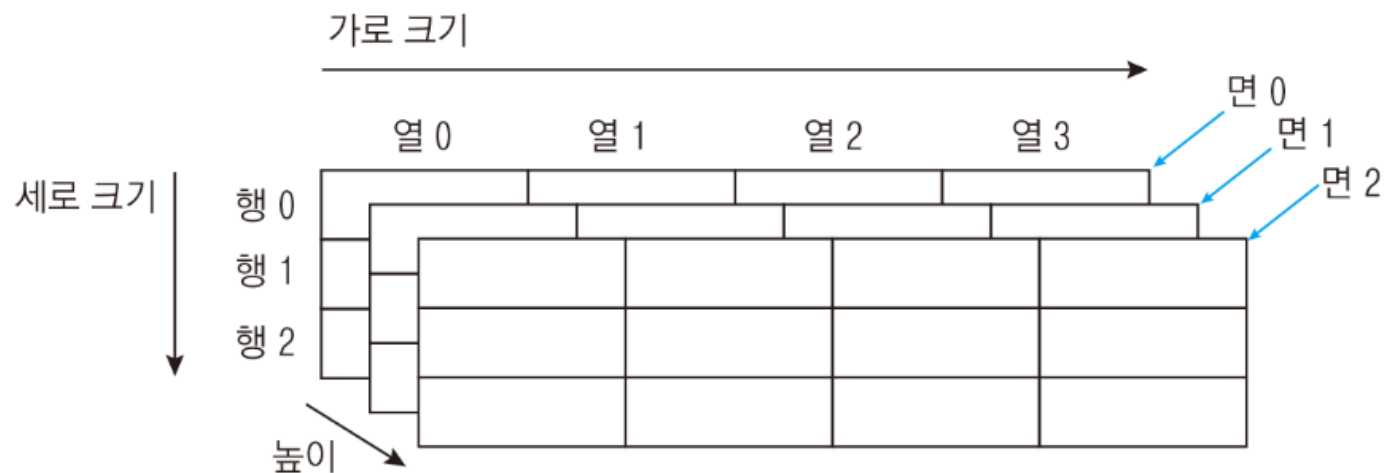
```
printf("%d\n", numArr[0][2][1]);    // 110  
numArr[1][1][2] = 0;               // 요소에 값 저장
```



3차원 배열

▶ 3차원 배열

- ▶ 3차원 배열의 높이(깊이) : $\text{sizeof}(\text{배열}) / \text{sizeof}(\text{배열}[0])$
 - ▷ 배열이 차지하는 전체공간을 면의 크기로 나눔
- ▶ 3차원 배열의 세로 크기 : $\text{sizeof}(\text{배열}[0]) / \text{sizeof}(\text{배열}[0][0])$
 - ▷ 한 면의 크기를 가로 한 줄의 크기로 나눠줌
- ▶ 3차원 배열의 가로 크기 : $\text{sizeof}(\text{배열}[0][0]) / \text{sizeof}(\text{자료형})$
 - ▷ 가로 한 줄의 크기를 요소의 크기로 나눠줌
- ▶ 3차원 배열을 포인터에 저장
 - ▷ 자료형 (*포인터이름)[세로크기][가로크기]
 - ▷ `int numArr[2][3][4] = {0,};`
 - ▷ `int (*numPtr)[3][4] = numArr;`



실습예제 05

▶ 2차원 배열

▶ 다음 소스 코드를 완성하여 정사각행렬의 주대각선 성분이 출력되게 만드시오.

▷ 주대각선 성분은 왼쪽 위부터 오른쪽 아래까지 이어지는 대각선에 위치한 값을 의미

```
#include <stdio.h>
int main(){
    int matrix[8][8] = {
        { 1, 2, 3, 4, 5, 6, 7, 8 },
        { 9, 10, 11, 12, 13, 14, 15, 16 },
        { 17, 18, 19, 20, 21, 22, 23, 24 },
        { 25, 26, 27, 28, 29, 30, 31, 32 },
        { 33, 34, 35, 36, 37, 38, 39, 40 },
        { 41, 42, 43, 44, 45, 46, 47, 48 },
        { 49, 50, 51, 52, 53, 54, 55, 56 },
        { 57, 58, 59, 60, 61, 62, 63, 64 }
    };

    _____
    ...
    _____

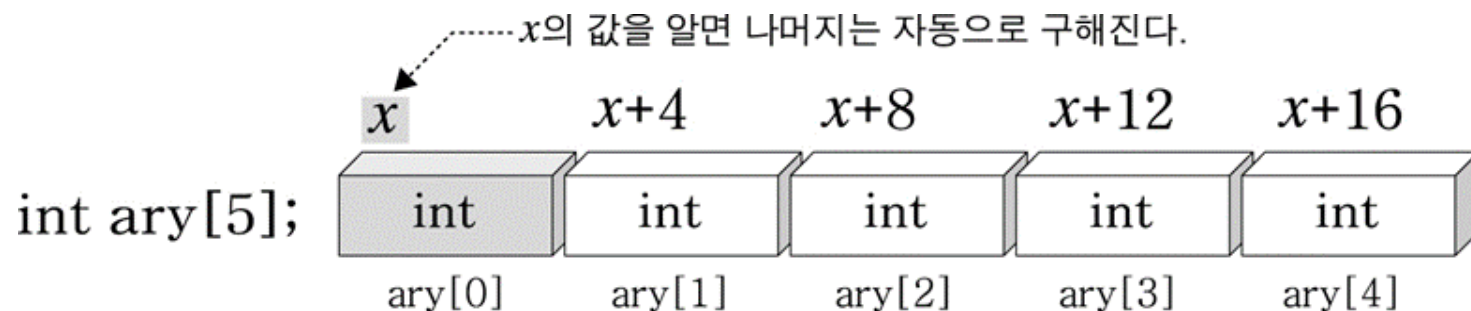
    return 0;
}
```

1 10 19 28 37 46 55 64

배열과 포인터

▶ 배열의 주소

- ▶ 배열을 선언할 때 사용한 배열명은 배열의 시작 주소를 갖고 있음
- ▶ 배열은 연속된 메모리 공간에 할당되어 있기 때문에 첫 번째 기억 공간을 알면 나머지 기억공간들의 위치도 자연스럽게 확인 가능
 - ▶ 결국 배열의 모든 기억 공간은 첫 번째 배열 요소의 시작 주소만 알면 참조 가능



```
int ary[5] = {10, 20, 30, 40, 50};
int i;
for(i=0; i < sizeof(ary)/sizeof(ary[0]); i++)
    printf("%d", ary[i]);
}
```

배열과 포인터

▶ 포인터 산술 연산

- ▶ 배열의 시작 주소를 알면 포인터로 모든 요소에 접근 가능
- ▶ 포인터(주소)에 정수 값을 더할 때는 포인터가 가리키는 자료형의 크기를 곱해서 더해줌
 - ▶ 예를 들어 int형 ary배열의 시작 주소에 4를 더하면 ary[4](ary +4)의 주소, 즉 다섯 번째 요소의 주소를 의미
 - ▶ int형의 경우 4바이트를 모두 읽어와야 정상적인 데이터를 확인 가능

포인터 + 정수값



포인터 + (정수값 * 포인터가 가리키는 자료형의 크기)

$\&\text{ary}[0] + 4 = \&\text{ary}[0] + (4 * \text{sizeof}(\text{int})) = 36 + 16 = 52\text{번지}$

```
int ary[5] = {10, 20, 30, 40, 50};
int * ptr = ary
int i;
for(i=0; i < sizeof(ary)/sizeof(ary[0]); i++)
    printf("%d", *(ptr + i));
}
```

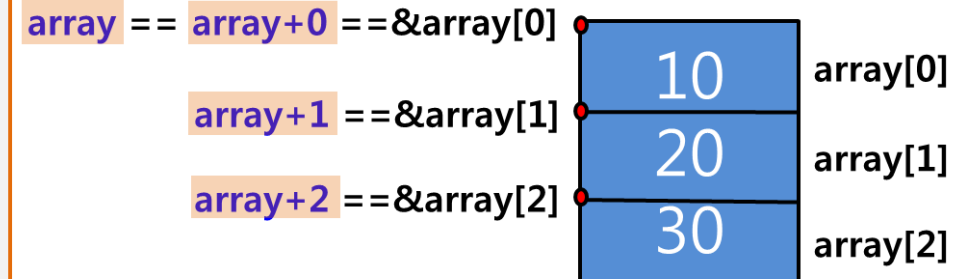
배열과 포인터

▶ 배열명으로 배열의 요소 접근

▶ 배열명을 이용하여 포인터의 기본 접근 방식으로 요소의 정보에 접근

```
#include <stdio.h>
int main(void)
{
    int array[3]={10, 20, 30};

    printf("%x %x %x \n", array, array+0, &array[0]);
    printf("%x %x \n", array+1, &array[1]);
    printf("%x %x \n", array+2, &array[2]);
    printf("%d %d %d \n", sizeof(array), sizeof(array+0), sizeof(&array[0]));
    return 0;
}
```



```
39fab8 39fab8 39fab8
39fab0 39fab0
39fac0 39fac0
12 4 4
```


배열과 포인터

▶ 포인터 변수를 통한 1차원 배열 요소의 주소 접근

▶ 배열의 시작 주소를 저장

```
#include <stdio.h>
int main(void)
{
    int array[3]={10, 20, 30};
    int* p=NULL;

    p=array; // p=&array[0];

    printf("%x %x %x \n", p, p+0, &p[0]);
    printf("%x %x \n",      p+1, &p[1]);
    printf("%x %x \n",      p+2, &p[2]);
    return 0;
}
```

```
daf6ec daf6ec daf6ec
daf6f0 daf6f0
daf6f4 daf6f4
```

배열과 포인터

▶ 포인터 변수를 통한 1차원 배열 요소의 값 접근

▶ *연산자 사용

```
#include <stdio.h>
int main(void)
{
    int array[3]={10, 20, 30};
    int* p=NULL;
    p=array;          // p=&array[0];

    // 주소에 *연산자를 붙임
    printf("%d %d %d \n", *p, *(p+0), *&p[0]); // *&는 서로 상쇄
    printf("%d %d \n",      *(p+1), *&p[1]); // *&는 서로 상쇄
    printf("%d %d \n",      *(p+2), *&p[2]); // *&는 서로 상쇄
    return 0;
}
```

```
10 10 10
20 20
30 30
```

배열과 포인터

▶ 배열과 포인터의 관계

- ▶ 배열은 [] 연산자를 사용하여 저장된 값을 표현하고 포인터는 *연산자를 이용하여 표현
- ▶ 배열과 포인터는 표기법을 서로 바꿔 사용 가능 `array[i] == *(array+i)`
- ▶ 배열명과 포인터 변수의 관계

구분	사용 예	기능
배열명	<code>int ary[3];</code> <code>ary == &ary[0];</code>	배열명은 첫 번째 요소의 주소
배열명 + 정수	<code>int ary[3];</code> <code>ary + 1;</code>	가리키는 자료형의 크기를 곱해서 더한다. <code>ary + (1 * sizeof(*ary))</code>
배열명과 포인터는 같다.	<code>int ary[3];</code> <code>int *pa = ary;</code> <code>pa[1] = 10;</code>	포인터가 배열명을 저장하면 배열명처럼 쓸 수 있다. 두 번째 배열 요소에 10 대입
배열명과 포인터는 다르다.	<code>ary++;</code> → (×) <code>pa++;</code> → (○)	배열명은 상수이므로 그 값을 바꿀 수 없지만 포인터는 가능하다.

배열과 포인터

▶ 포인터 변수와 배열명 차이

▶ 배열명은 상수이므로 저장된 배열의 주소 값을 변경할 수 없음

▶ pa는 포인터 변수, ary는 배열명일 경우

pa = pa + 1	가능	// pa에 1을 더하여 다시 pa에 저장할 수 있다.
ary = ary + 1	불가능	// ary에 1을 더하는 것은 가능하나 그 값을 다시 저장할 수 없다.

▶ sizeof 함수 사용 결과의 차이

▶ 배열명에 사용시 배열 전체의 크기

▶ 포인터에 사용하면 포인터 하나의 크기

▶ 배열명을 포인터에 저장하면 포인터로 배열 전체 크기 확인 불가

```
int ary[3];  
int *pa = ary;
```

sizeof (ary)	12바이트	// 배열 전체 크기
sizeof (pa)	4바이트	// 포인터 하나의 크기

배열과 포인터

▶ 배열의 특징

▶ 배열은 포인터처럼 작동하지만 포인터와 동일하지 않음

▷ 문법적으로 포인터는 다른 변수의 주소를 저장 가능하지만 배열은 컴파일러가 제공하는 메모리 그룹화 기술

▷ 배열명은 포인터 변수처럼 보이지만 내부는 상수화된 주소 - 배열의 시작 주소를 변경할 수 없음

```
int main() {  
    int num = 100;  
    int* ptr = &num;  
    int ary[10] = { 0 };  
    ptr = ary;  
    //ary = ptr;    //error  
    printf("%x\n", ary);  
    printf("%x\n", &ary);  
    printf("%x\n", ptr);  
    printf("%x\n", &ptr);  
    return 0;  
}
```

```
83f738  
83f738  
83f738  
83f768
```

실습예제 06

▶ 배열과 포인터

▶ 배열의 평균값 구하기

▷ 다음 배열의 평균값을 구하여 출력하는 프로그램을 작성하시오.

▷ `double ary[] = {1.5, 20.1, 16.4, 2.3, 3.5}` //5개의 요소

▷ 코드에서 배열 요소를 참조할 경우에는 배열 명에 정수 값을 더하는 포인터 표현을 사용

▷ 평균은 소수점 둘째 자리까지 출력

▷ 실행 결과

평균값 : 8.76

실습예제 07

▶ 배열과 포인터

▶ 배열의 값들을 거꾸로 출력하는 프로그램 작성

▷ 포인터 변수를 사용하여 다음 배열의 값들을 거꾸로 출력하는 프로그램을 작성하시오.

▷ `double ary[] = {1.5, 20.1, 16.4, 2.3, 3.5}` //5개의 요소

▷ 코드에서 배열 요소를 참조할 경우에는 배열 명에 정수 값을 더하는 포인터 표현을 사용

▷ 실행 결과

```
3.5  2.3  16.4  20.1  1.5↵
```

실습예제 08

▶ 배열과 포인터

▶ 배열에서 최소값 구하기

▷ 다음 배열에서 최소값을 구하고 출력하는 프로그램을 작성하시오.

▷ `double ary[] = {1.5, 20.1, 16.4, 2.3, 3.5}` //5개의 요소

▷ 배열의 값을 참조하여 최소값을 리턴하는 함수를 만들고 호출하여 작성

▷ `double findMin(double * ary, int count)`

▷ 실행 결과

배열의 최소값 : 1.5

실습예제 09

▶ 배열과 포인터

▶ 5개의 정수를 키보드로 입력 받아 오름차순으로 정렬하여 출력하는 프로그램을 작성하시오.

```
5개의 정수 입력 : 4 2 5 3 1  
1 2 3 4 5
```

포인터를 이용한 동적 배열

▶ 포인터 사용하기

- ▶ 지금까지 사용한 배열은 크기가 고정된 배열
- ▶ `int numArr[10];`처럼 크기를 지정해서 생성
- ▶ 가변 길이 배열 (Variable-Length Array, VLA) 가변 길이 배열은 Visual Studio에서 지원하지 않음

```
#include <stdio.h>

int main()
{
    int size;

    scanf_s("%d", &size); // 배열의 크기를 입력받음

    int numArr[size];      // GCC에서는 사용 가능, Visual Studio 2015에서는 컴파일 에러

    return 0;
}
```

```
error C2057: 상수 식이 필요합니다.
error C2466: 상수 크기 0의 배열을 할당할 수 없습니다.
error C2133: 'numArr': 알 수 없는 크기입니다.
```

포인터를 이용한 동적 배열

▶ 포인터를 이용한 동적 배열

▶ 포인터를 선언하고 메모리를 할당한 뒤 할당된 메모리를 배열처럼 사용

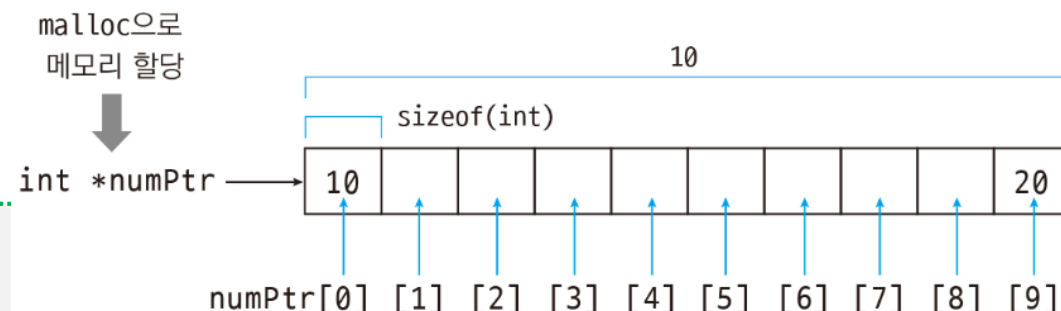
▶ 자료형 *포인터이름 = malloc(sizeof(자료형) * 크기);

▷ 포인터에 malloc 함수로 메모리 할당

▶ 포인터[인덱스] 방식으로 접근

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

```
int main(){
    int *numPtr = malloc(sizeof(int) * 10);    // int 10개 크기만큼 동적 메모리 할당
    numPtr[0] = 10;    // 배열처럼 인덱스로 접근하여 값 할당
    numPtr[9] = 20;    // 배열처럼 인덱스로 접근하여 값 할당
    printf("%d\n", numPtr[0]);    // 배열처럼 인덱스로 접근하여 값 출력
    printf("%d\n", numPtr[9]);    // 배열처럼 인덱스로 접근하여 값 출력
    free(numPtr);    // 동적으로 할당한 메모리 해제
    return 0;
}
```



10
20

포인터를 이용한 동적 배열

▶ 입력된 크기만큼 동적으로 메모리를 할당하여 배열처럼 사용

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

int main(){
    int size;
    scanf("%d", &size);
    int *numPtr = malloc(sizeof(int) * size);    // (int 크기 * 입력받은 크기)만큼 동적 메모리 할당
    for (int i = 0; i < size; i++)    // 입력받은 크기만큼 반복
    {
        numPtr[i] = i;                // 인덱스로 접근하여 값 할당
    }
    for (int i = 0; i < size; i++)    // 입력받은 크기만큼 반복
    {
        printf("%d\n", numPtr[i]);    // 인덱스로 접근하여 값 출력
    }
    free(numPtr);    // 동적으로 할당한 메모리 해제
    return 0;
}
```

5 #입력
0
1
2
3
4

포인터를 이용한 동적 배열

▶ 포인터에 할당된 메모리를 2차원 배열처럼 사용하는 방법

```
#include <stdio.h>
#include <stdlib.h>    // malloc, free 함수가 선언된 헤더 파일
int main(){
    int **m = malloc(sizeof(int *) * 3);    // 이중 포인터에(int 포인터 크기 * 세로 크기)만큼 동적 메모리 할당. 배열의 세로
    for (int i = 0; i < 3; i++)              // 세로 크기만큼 반복
    {
        m[i] = malloc(sizeof(int) * 4);    // (int 크기 * 가로 크기)만큼 동적 메모리 할당.
        // 배열의 가로

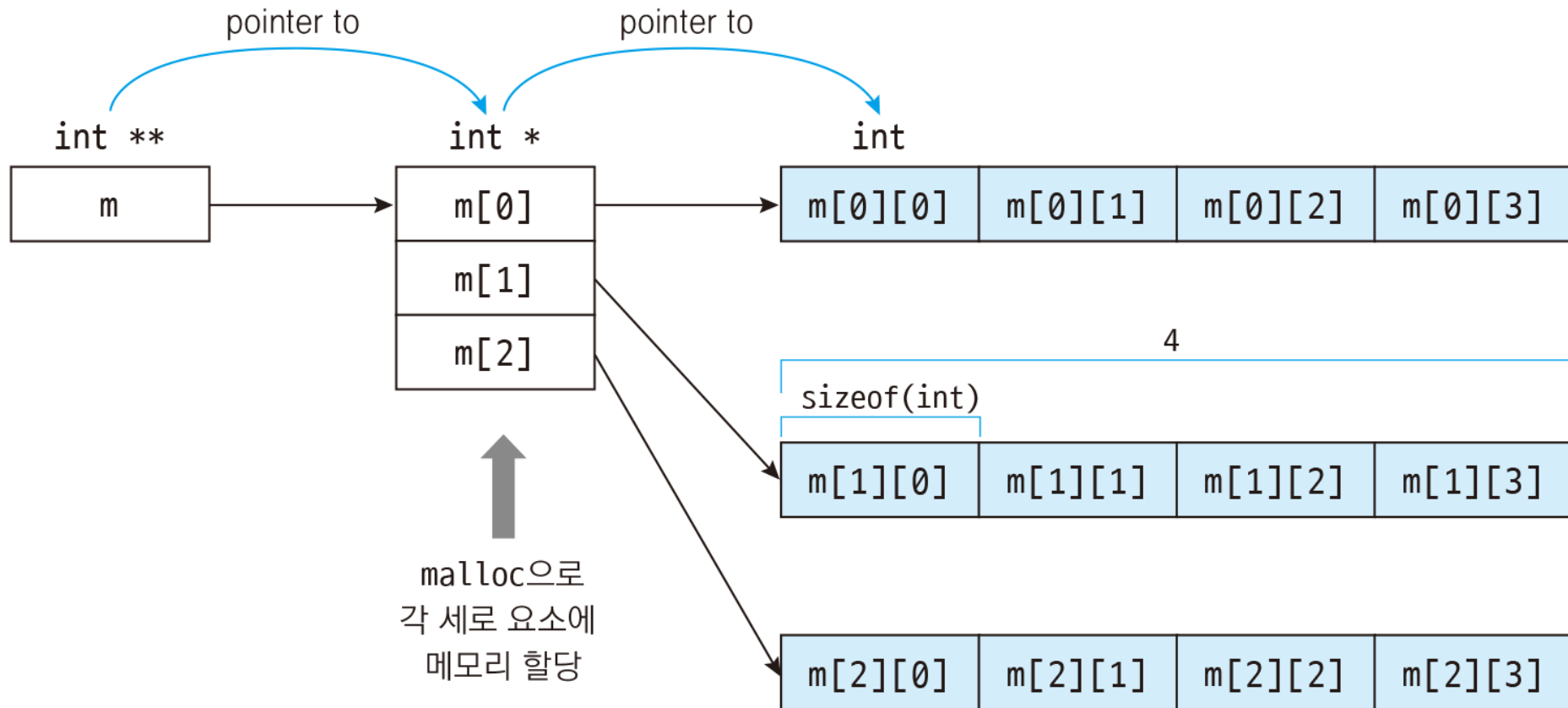
        m[0][0] = 1;    // 세로 인덱스 0, 가로 인덱스 0인 요소에 값 할당
        m[2][0] = 5;    // 세로 인덱스 2, 가로 인덱스 0인 요소에 값 할당
        m[2][3] = 2;    // 세로 인덱스 2, 가로 인덱스 3인 요소에 값 할당
        printf("%d\n", m[0][0]);    // 1: 세로 인덱스 0, 가로 인덱스 0인 요소의 값 출력
        printf("%d\n", m[2][0]);    // 5: 세로 인덱스 2, 가로 인덱스 0인 요소의 값 출력
        printf("%d\n", m[2][3]);    // 2: 세로 인덱스 2, 가로 인덱스 3인 요소의 값 출력
        for (int i = 0; i < 3; i++)    // 세로 크기만큼 반복
        {
            free(m[i]);                // 2차원 배열의 가로 공간 메모리 해제
        }
        free(m);    // 2차원 배열의 세로 공간 메모리 해제
        return 0;
    }
```

1
5
2

포인터를 이용한 동적 배열

▶ 포인터에 할당된 메모리를 2차원 배열처럼 사용하는 방법

▶ 메모리 할당 과정



포인터를 이용한 동적 배열

▶ 입력한 크기만큼 메모리를 할당하여 포인터를 2차원 배열처럼 사용

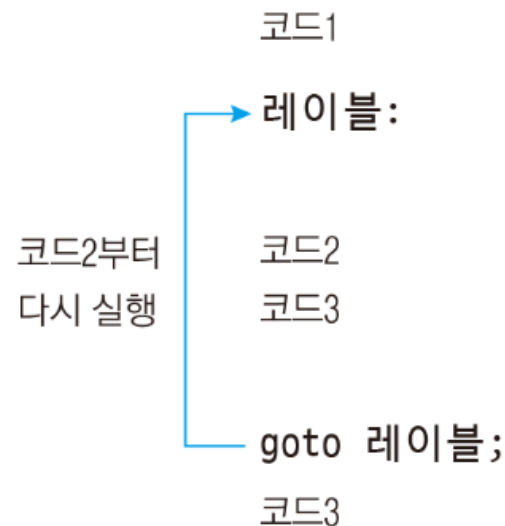
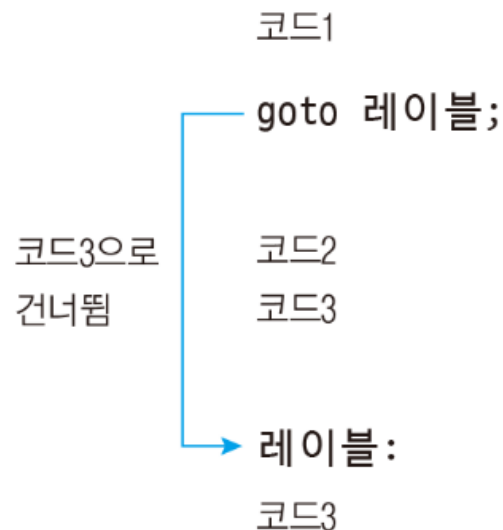
```
int main(){
    int row, col;
    scanf("%d %d", &row, &col);
    int **m = malloc(sizeof(int *) * row);    // 이중 포인터에 (int 포인터 크기 * row)만큼 동적 메모리 할당. 배열의 세로
    for (int i = 0; i < row; i++){            // 세로 크기만큼 반복
        m[i] = malloc(sizeof(int) * col);    // (int의 크기 * col)만큼 동적 메모리 할당. 배열의 가로
    }
    for (int i = 0; i < row; i++){            // 세로 크기만큼 반복
        for (int j = 0; j < col; j++){        // 가로 크기만큼 반복
            m[i][j] = i + j;                // 2차원 배열의 각 요소에 i + j 값을 할당
        }
    }
    for (int i = 0; i < row; i++){            // 세로 크기만큼 반복
        for (int j = 0; j < col; j++){        // 가로 크기만큼 반복
            printf("%d ", m[i][j]);          // 2차원 배열의 인덱스에 반복문의 변수 i, j를 지정
        }
        printf("\n");                        // 가로 요소를 출력한 뒤 다음 줄로 넘어감
    }
    for (int i = 0; i < row; i++){            // 세로 크기만큼 반복
        free(m[i]);                          // 2차원 배열의 가로 공간 메모리 해제
    }
    free(m);    // 2차원 배열의 세로 공간 메모리 해제
    return 0;
}
```

4	5	(입력)		
0	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7

goto문

▶ goto문

- ▶ 프로그램을 작성할 때 중간의 코드는 무시하고 건너 뛸 때 제어문 goto를 사용
- ▶ goto를 적절히 활용하면 중복되는 코드를 없애고, 코드를 간결하게 만들 수 있음
- ▶ 에러 처리에 매우 유용함(리눅스 커널에서도 자주 사용됨)
- ▶ goto를 남발하는 경우에는 스파게티 코드 문제가 될 수 있음
 - ▷ goto를 과도하게 사용한 코드
 - ▷ 코드가 스파게티 면발처럼 꼬였다고 해서 붙여진 이름
 - ▷ 가독성이 떨어지고 유지보수가 어려움
 - ▷ 이러한 이유 때문에 대체로 goto문 사용을 권장하지 않음



goto문

▶ goto문 레이블 사용

▶ goto문은 레이블을 지정하여 사용

▶ 레이블은 변수 이름 규칙과 동일하며 끝에 콜론(:)을 붙임

▶ goto 레이블

▶ 레이블 :

```
if (num1 == 1)           // num1이 1이면
    goto ONE;           // 레이블 ONE으로 즉시 이동
else if (num1 == 2)      // num1이 2이면
    goto TWO;           // 레이블 TWO로 즉시 이동
else                     // 1도 아니고 2도 아니면
    goto EXIT;          // 레이블 EXIT로 즉시 이동
```

```
ONE:    // 레이블 ONE
    printf("1입니다.\n");
    goto EXIT; // 레이블 EXIT로 즉시 이동
TWO:    // 레이블 TWO
    printf("2입니다.\n");
    goto EXIT; // 레이블 EXIT로 즉시 이동
EXIT:   // 레이블 EXIT
    return 0;
```

▶ goto문 예제

```
int main()
{
    int num1;
    while (1) {
        printf("이동할 위치 번호 : ");
        scanf_s("%d", &num1);

        if (num1 == 1)           // num1이 1이면
            goto ONE;           // 레이블 ONE으로 즉시 이동
        else if (num1 == 2)      // num1이 2이면
            goto TWO;           // 레이블 TWO로 즉시 이동
        else                    // 1도 아니고 2도 아니면
            goto EXIT;          // 레이블 EXIT로 즉시 이동
    }
    ONE:    // 레이블 ONE
        printf("1입니다.\n");
        goto EXIT; // 레이블 EXIT로 즉시 이동
    TWO:    // 레이블 TWO
        printf("2입니다.\n");
        goto EXIT; // 레이블 EXIT로 즉시 이동
    EXIT:   // 레이블 EXIT
        return 0;
}
```

▶ goto문을 if-else if문으로 변경

```
#include <stdio.h>

int main()
{
    int num1;

    scanf("%d", &num1);

    if (num1 == 1)          // num1이 1이면
        printf("1입니다.\n");
    else if (num1 == 2)     // num1이 2이면
        printf("2입니다.\n");

    return 0;
}
```

▶ 중첩 루프 빠져 나오는 코드

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
    int num1 = 0;
    bool exitOuterLoop = false;    // 바깥쪽 루프를 빠져나올지 결정하는 변수
    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            if (num1 == 20)          // num1이 20이라면
            {
                exitOuterLoop = true;    // 바깥쪽 루프도 빠져나가겠음
                break;                  // 안쪽 루프를 끝냄
            }
            num1++;
        }
        if (exitOuterLoop == true)    // 바깥쪽 루프도 빠져나오겠다고 결정했으면
            break;                    // 바깥쪽 루프를 끝냄
    }
    printf("%d\n", num1);    // 20
    return 0;
}
```

goto문

▶ goto문을 이용하여 중첩 루프 빠져 나오기

▶ 한번에 빠져나올 수 있음

```
#include <stdio.h>

int main()
{
    int num1 = 0;
    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            if (num1 == 20)    // num1이 20이라면
                goto EXIT;    // 레이블 EXIT로 즉시 이동
            num1++;
        }
    }
EXIT:    // 레이블 EXIT
    printf("%d\n", num1);    // 20
    return 0;
}
```

▶ 에러 처리 패턴

▶ 자가 주택을 소유한 30대 남성을 구분하는 코드

```
int main()
{
    int gender;    // 성별: 남자 1, 여자 2
    int age;       // 나이
    int isOwner;   // 주택 소유 여부: 자가 1, 임대 0

    scanf("%d %d %d", &gender, &age, &isOwner);

    if (gender == 2)    // 여자라면
    {
        printf("조건에 맞지 않음.\n");    // 중복 코드
        return 0;                        // 프로그램 종료
    }
}
```

```
if (age < 30)    // 30세 미만이라면
{
    printf("조건에 맞지 않음.\n");    // 중복 코드
    return 0;                        // 프로그램 종료
}

if (isOwner == 0)    // 전월세라면
{
    printf("조건에 맞지 않음.\n");    // 중복 코드
    return 0;                        // 프로그램 종료
}
printf("조건에 맞음");
return 0;    // 프로그램 종료
}
```

▶ goto와 에러 처리 패턴

```
int main()
{
    int gender;    // 성별: 남자 1, 여자 2
    int age;       // 나이
    int isOwner;   // 주택 소유 여부: 자가 1, 임대 0
    scanf("%d %d %d", &gender, &age, &isOwner);
    printf("안녕하세요.\n");
    printf("문을 연다.\n");
    if (gender == 2)
        goto EXIT;    // 에러가 발생했으므로 EXIT로 이동
    if (age < 30)
        goto EXIT;    // 에러가 발생했으므로 EXIT로 이동
    if (isOwner == 0)
        goto EXIT;    // 에러가 발생했으므로 EXIT로 이동
    printf("조건에 맞음");
EXIT:
    printf("조건에 맞지 않음.\n");    // 에러 처리 코드를 한 번만 사용함
    return 0;    // 프로그램 종료
}
```

실습예제 10

▶ switch에서 반복문 빠져나오기

▶ 다음 소스 코드를 완성하여 아래와 같이 출력되게 만드세요.

```
int main()
{
    int num1 = 1;

    for (int i = 0; i < 10; i++)
    {
        switch (num1)
        {
            case 1:
                printf("1입니다.\n");
                ① _____
            default:
                break;
        }
    }

    ② _____
    return 0;
}
```

1입니다.

실습예제 11

▶ 중첩 루프 빠져나오기

▶ 다음 소스 코드의 실행 결과가 아래와 같이 출력되도록 완성하시오.

```
#include <stdio.h>
int main()
{
    int num1 = 0;
    for (int i = 0; i < 5; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            if (num1 == 10)
                goto _____
            num1++;
        }
    }
    EXIT1:
        printf("100\n");
    EXIT2:
        printf("200\n");
    EXIT3:
        printf("300\n");
    return 0;
}
```

```
200
300
```

Q & A
