

프로그래밍언어

7. 포인터 (Pointer)와 동적 배열



교수 김 영 탁

영남대학교 기계IT대학 정보통신공학과

(Tel : +82-53-810-2497; Fax : +82-53-810-4742

<http://antl.yu.ac.kr/>; E-mail : ytkim@yu.ac.kr)

Outline

- ◆ 실행중인 프로그램의 메모리 맵, 함수와 변수의 주소
- ◆ 포인터 (pointer)란 ?
- ◆ 간접 참조 연산자 (&, *)
- ◆ 포인터 연산
- ◆ 포인터와 배열
- ◆ 함수 호출에서의 포인터 인수 (Call-by-Pointer)
- ◆ 포인터의 배열 (array of pointers)
- ◆ 배열의 동적 생성 (dynamic array)
- ◆ 2차원 배열의 동적 생성



**실행중인 프로그램의 Memory Map,
함수와 변수의 주소**

실행중인 프로그램의 함수와 변수의 주소

◆ 실행중인 프로그램의 주소 정보 파악

- 함수들의 주소: main(), subroutine() 등
- auto 지역 변수의 주소
- static 지역 변수의 주소
- 전역 변수 (global variable)의 주소
- 동적 메모리 할당 (dynamic memory allocation) 블록의 주소



실행중인 프로그램의 함수와 변수의 주소

```
/* PointerMemoryAddr.c (1) */

#include <stdio.h>
#include <stdlib.h>
int global_var;
#define ARRAY_SIZE 1024
void recursive_call(int level);
void main()
{
    int local_x;
    static int static_local;

    printf("Address of main() function: %p\n", main);
    printf("Address of global var: %p\n", &global_var);
    printf("Address of local var in main(): %p\n", &local_x);
    printf("Address of static local variable: %p\n", &static_local);

    recursive_call(0);
}
```



```

/* PointerMemoryAddr.c (2) */
void recursive_call(int level)
{
    int auto_y;
    static int static_y;
    int array[ARRAY_SIZE];
    int *dynamicArray = NULL;

    if (level >= 20)
        return;
    dynamicArray = (int *)calloc(ARRAY_SIZE , sizeof(int));
    printf("Addr of local variables (in recursive level %3d): auto_y: %p,
           static_y: %p, dynamicArray: %p\n", level, &auto_y, &static_y, dynamicArray);
    recursive_call(level + 1);
    free(dynamicArray);
}

```

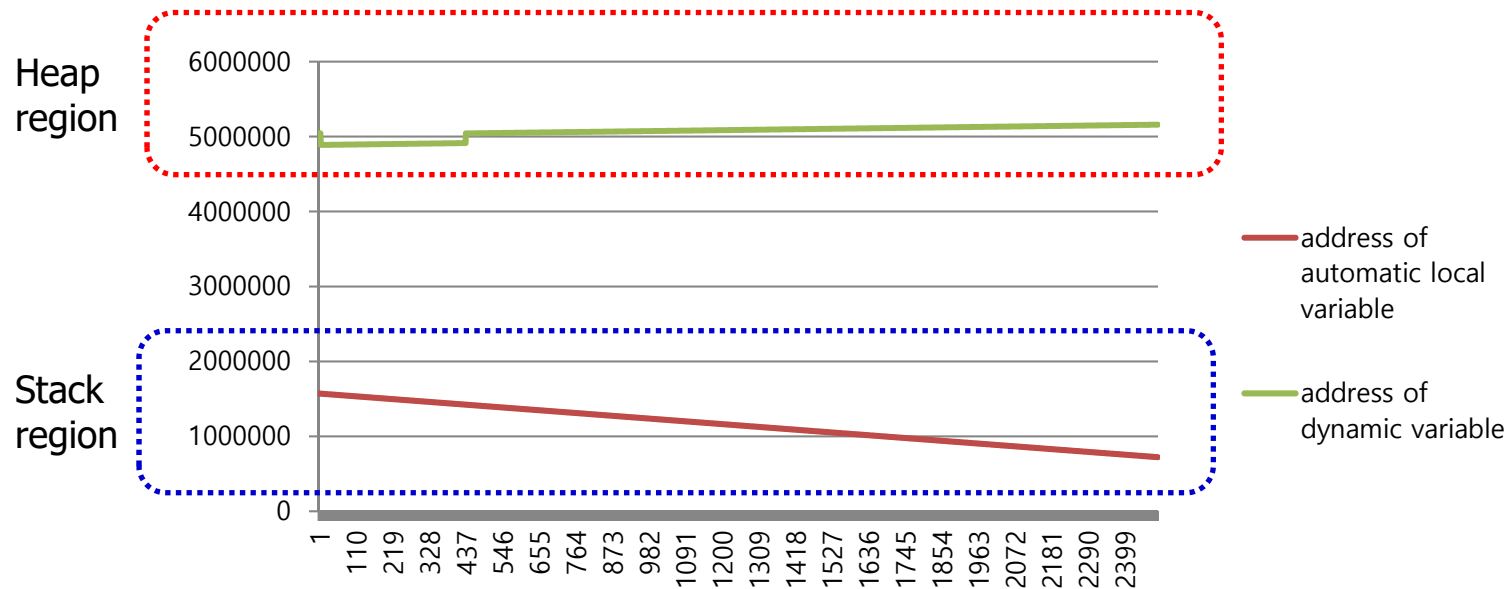
```

Address of main() function: 0131114F
Address of global var: 01318130
Address of local var in main(): 0044FB98
Address of static local variable: 01318134
Addr of local variables (in recursive level 0): auto_y: 0044FAB4, static_y: 01318138, dynamicArray: 008B6480
Addr of local variables (in recursive level 1): auto_y: 0044E9B0, static_y: 01318138, dynamicArray: 008B7480
Addr of local variables (in recursive level 2): auto_y: 0044D8AC, static_y: 01318138, dynamicArray: 008B84E0
Addr of local variables (in recursive level 3): auto_y: 0044C7A8, static_y: 01318138, dynamicArray: 008B9510
Addr of local variables (in recursive level 4): auto_y: 0044B6A4, static_y: 01318138, dynamicArray: 008BA540
Addr of local variables (in recursive level 5): auto_y: 0044A5A0, static_y: 01318138, dynamicArray: 008BB570
Addr of local variables (in recursive level 6): auto_y: 0044949C, static_y: 01318138, dynamicArray: 008BC5A0
Addr of local variables (in recursive level 7): auto_y: 00448398, static_y: 01318138, dynamicArray: 008BD5D0
Addr of local variables (in recursive level 8): auto_y: 00447294, static_y: 01318138, dynamicArray: 008BE600
Addr of local variables (in recursive level 9): auto_y: 00446190, static_y: 01318138, dynamicArray: 008BF630
Addr of local variables (in recursive level 10): auto_y: 0044508C, static_y: 01318138, dynamicArray: 008C0660
Addr of local variables (in recursive level 11): auto_y: 00443F88, static_y: 01318138, dynamicArray: 008C1690
Addr of local variables (in recursive level 12): auto_y: 00442E84, static_y: 01318138, dynamicArray: 008C26C0
Addr of local variables (in recursive level 13): auto_y: 00441D80, static_y: 01318138, dynamicArray: 008C36F0
Addr of local variables (in recursive level 14): auto_y: 00440C7C, static_y: 01318138, dynamicArray: 008C4720
Addr of local variables (in recursive level 15): auto_y: 0043FB78, static_y: 01318138, dynamicArray: 008C5750
Addr of local variables (in recursive level 16): auto_y: 0043EA74, static_y: 01318138, dynamicArray: 008C6798
Addr of local variables (in recursive level 17): auto_y: 0043D970, static_y: 01318138, dynamicArray: 008C78A0
Addr of local variables (in recursive level 18): auto_y: 0043C86C, static_y: 01318138, dynamicArray: 008C89A8
Addr of local variables (in recursive level 19): auto_y: 0043B768, static_y: 01318138, dynamicArray: 008C9AB0
계속하려면 아무 키나 누르십시오 . . .

```



◆ Address of local variables and dynamic variables in recursive function calls



◆ According to Operating System (OS) at each computer system, the actual memory management strategies are different !!

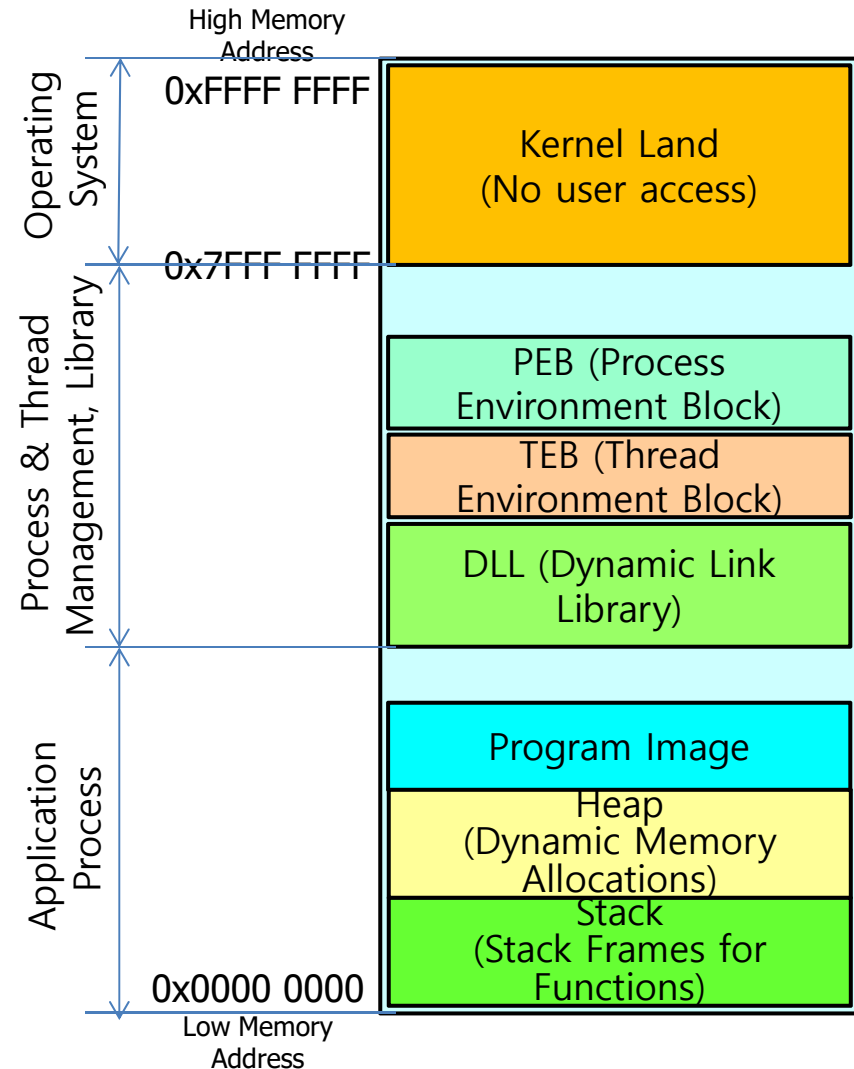
- Memory management is **UNIX/Linux** is different from the memory managements in **MS-Windows**



Process Memory Map (Example)

◆ 프로그램의 실행

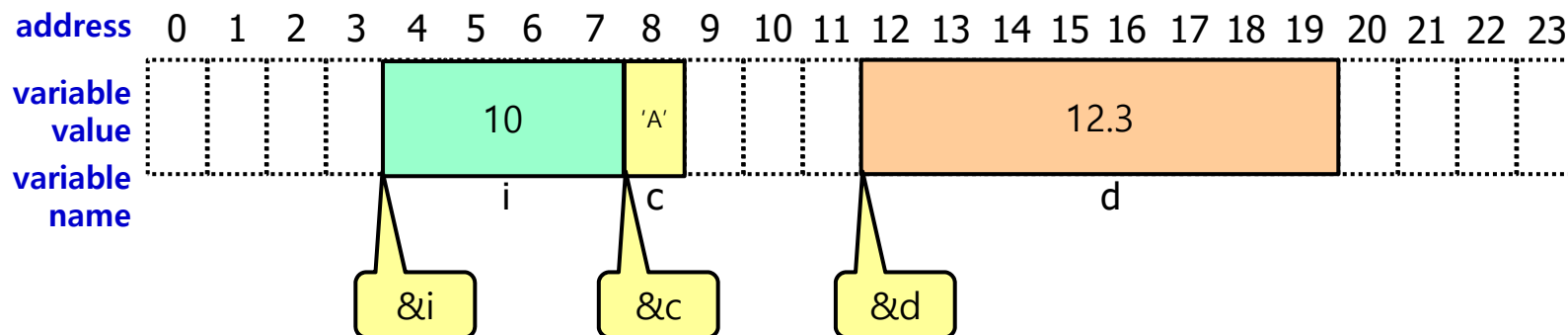
- Windows / Desktop PC에서 프로그램 실행 (process) 당 4GByte의 메모리 할당
- 프로그램 실행에서 사용되는 메모리 주소는 32 비트로 표현 ($2^{32} = 4 \text{ Giga-Byte}$)
- 사용자 응용 프로그램의 지역 변수는 Stack에 할당
사용자 응용 프로그램의 동적 메모리 할당은 Heap 사용



변수, 변수 주소, 주소 연산자

- ◆ 변수의 크기에 따라서 차지하는 메모리 공간이 달라진다.
- ◆ **char**형 변수: 1바이트, **int**형 변수: 4바이트,...
- ◆ 변수의 주소를 계산하는 주소 연산자: **&**
- ◆ 변수 **d**의 주소: **&d**

```
int main(void)
{
    int i = 10;
    char c = 'A';
    double d = 12.3;
}
```



변수 (Variable)의 주소

```
/* Test_Pointer_and_Addr.c */  
#include <stdio.h>
```

```
int global_i = 20;  
char global_c = 'G';  
double global_d = 567.1234;
```

```
void main()
```

```
{  
    int i = 10;  
    char c = 'A';  
    double d = 123.456;  
  
    printf("Address of integer i: %p\n", &i);  
    printf("Address of character c: %p\n", &c);  
    printf("Address of double d: %p\n", &d);  
    printf("Address of global_i: %p\n", &global_i);  
    printf("Address of global_c: %p\n", &global_c);  
    printf("Address of global_d: %p\n", &global_d);  
}
```

```
Address of integer i: 0022FDC8  
Address of character c: 0022FDBF  
Address of double d: 0022FDAC  
Address of global_i: 00A18000  
Address of global_c: 00A18004  
Address of global_d: 00A18008  
계속하려면 아무 키나 누르십시오 . . .
```



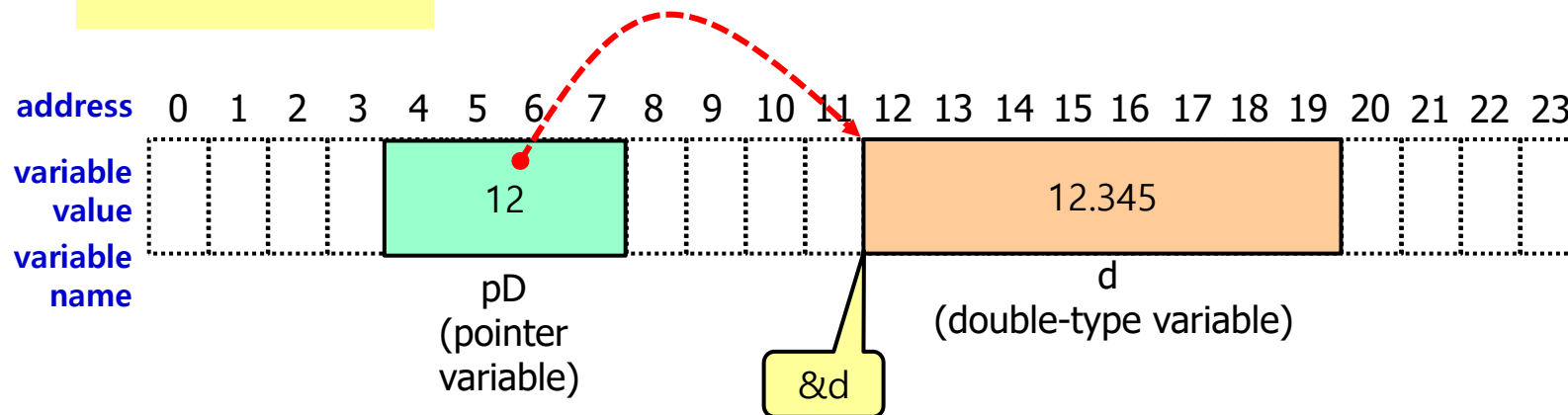
포인터 (Pointer)와 포인터 관련 연산자

포인터란?

◆ 포인터(pointer)

- 포인터는 변수 (variable)이며, 그 값으로 주소를 가지고, 값이 변경될 수 있다.
- 포인터는 가리키는 개체의 자료형 (data type)이 지정되며, 다른 자료형으로 설정하면 에러가 발생된다.

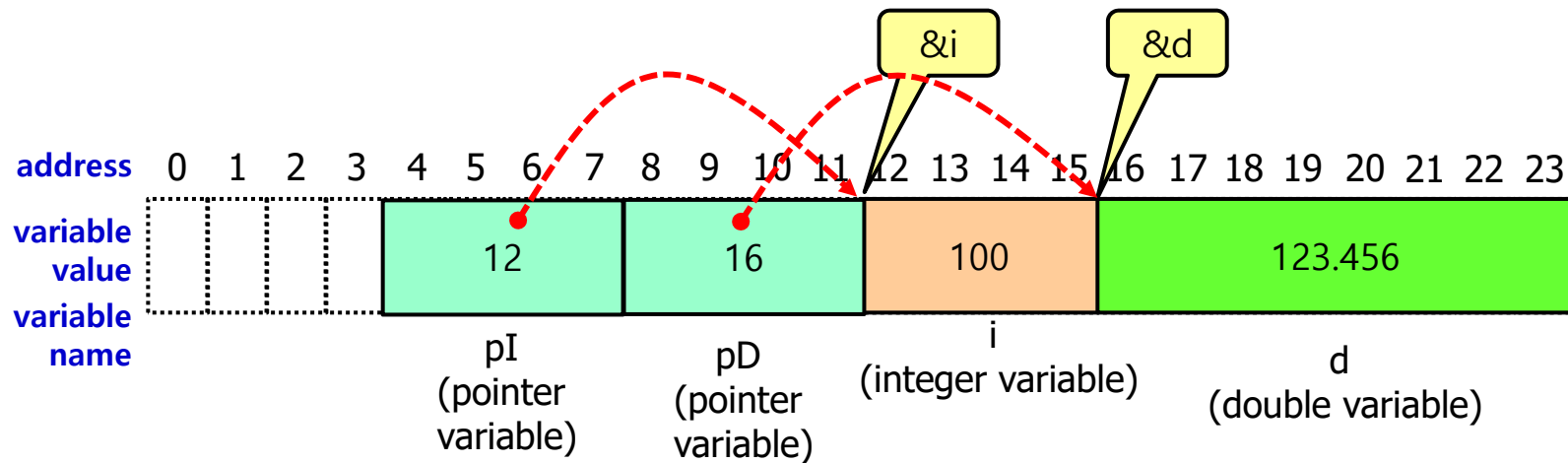
```
double d = 12.345;  
double *pD;  
  
pD = &d;
```



포인터 변수의 선언, 포인터 변수와 데이터 변수의 관계

```
int i = 100; // 정수형 데이터 변수  
double d = 123.456; // 더블형 데이터 변수  
int *pI = NULL; // 포인터 변수  
double *pD = NULL; // 포인터 변수
```

```
pI = &i; // 포인터 변수 pI의 값으로 정수형 데이터 변수 i의 주소 값을 대입  
pD = &d; // 포인터 변수 pD의 값으로 더블형 데이터 변수 d의 주소 값을 대입
```



포인터 연산자

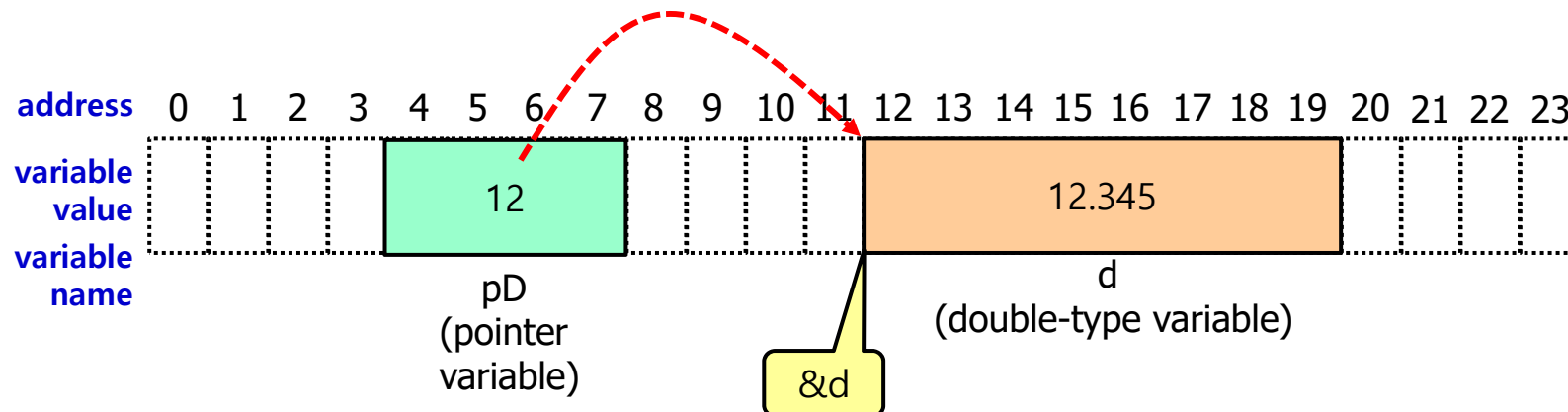
연산자의 분류	연산자	의미
포인터 관련 연산자 (Operators for pointer)	주소 연산자 &	변수의 주소를 찾아낼 때 사용
	간접참조 연산자 *	포인터가 가르키고 있는 곳의 값을 읽거나 쓸 때 사용
	[]	포인터를 배열의 이름처럼 사용하여, 동적 배열로 사용할 때



간접 참조 연산자

◆ 간접 참조 연산자 *: 포인터가 가리키는 곳의 값을 가져오거나 설정하는 연산자

```
double d;  
double *pD=&d;  
*pD = 12.345;    // pD가 가리키는 곳에 지정된 값을 저장  
printf("%lf", *pD); // pD가 가리키는 곳의 값을 가져옴
```

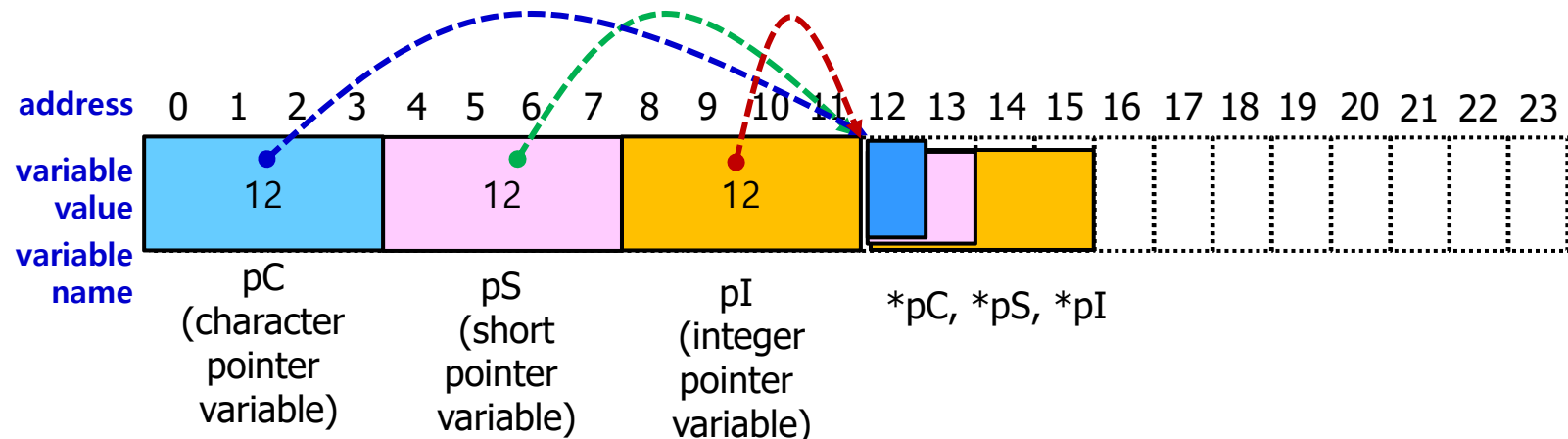


간접 참조 연산자의 해석

◆ 포인터 데이터 형에 따른 간접 참조 연산자

- 포인터가 지정하는 위치에서 포인터의 데이터 형에 따라 값을 읽어 들인다.

```
unsigned char data[16] = "012345678";  
char *pC = (char *)&data;    // 포인터 pC의 값으로 배열 data의 주소를 설정  
short *pS = (short *) &data; // 포인터 pS의 값으로 배열 data의 주소를 설정  
int *pI = (int *) &data;      // 포인터 pI의 값으로 배열 data의 주소를 설정  
  
char c = *pC;    // 포인터 pC가 가리키는 곳에서 문자 (1 byte)를 읽어옴  
short s = *pS;   // 포인터 pS가 가리키는 곳에서 문자 (2 byte)를 읽어옴  
int i = *pI;     // 포인터 pI가 가리키는 곳에서 정수 (4 bytes)를 읽어옴
```



void test_accessWithVariousPointerTypes()

```
{
    char data[17] = "0123456789ABCDEF";

    char *pC = (char *)&data;
    short *pS = (short *)&data;
    int *pI = (int *)&data;

    char c;
    short s;
    int i;
    unsigned char uc;

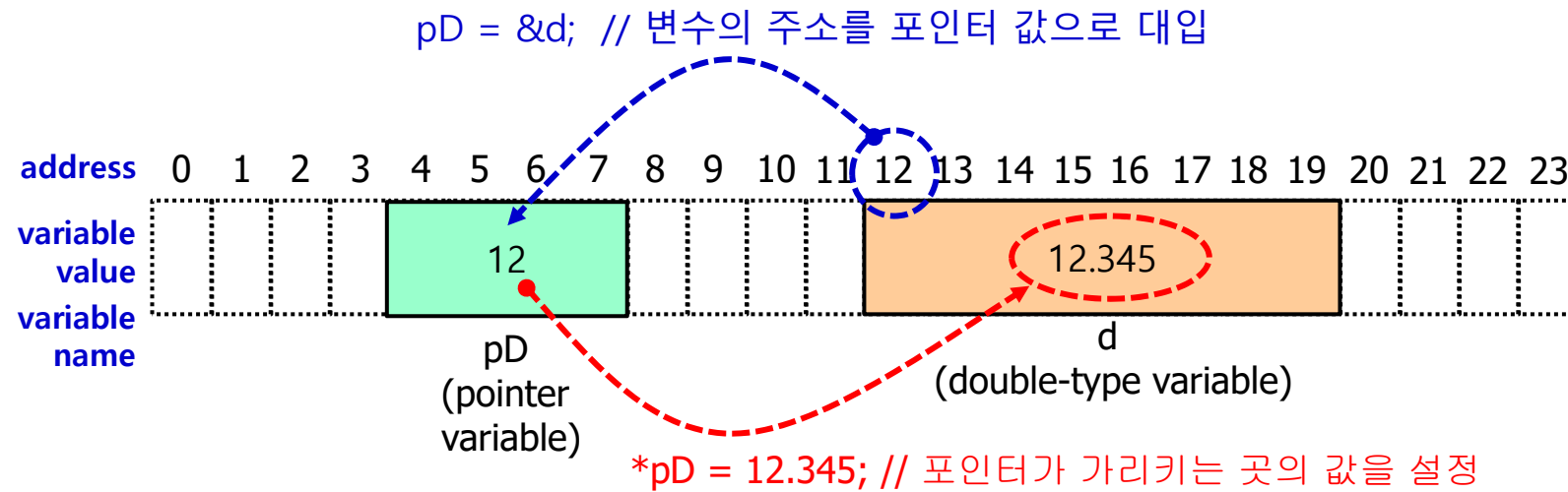
    c = *pC; // read 1 byte
    s = *pS; // read 2 bytes
    i = *pI; // read 4 bytes

    printf("data: %s (", data);
    for (int j = 0; j < strlen(data); j++)
    {
        uc = data[j];
        printf("%#01x ", uc);
    }
    printf("\n");
    printf("c (%#01x) = ", c); printOctet_Bits(c); printf("\n");
    printf("s (%#02x) = ", s); printShort_Bits(s); printf("\n");
    printf("i (%#04x) = ", i); printInt_Bits(i); printf("\n");
    printf("\n");
}
```

```
data: 0123456789ABCDEF (30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46 )
c (0x30) = 00110000
s (0x3130) = 00110001 00110000
i (0x33323130) = 00110011 00110010 00110001 00110000
계속하려면 아무 키나 누르십시오 . . .
```



& 연산자와 * 연산자



포인터 예제 #1

```
int test_IndirectAccessUsingPointers(void)
{
    double x = 10.0, y = 20.0;
    double *p = NULL;

    p = &x;
    printf("\n p = %p\n", p);
    printf("*p = %10.3lf\n", *p);
    printf(" x = %10.3lf\n", x);
    *p = 50.123;
    printf("After *p = 50.123; ==> \n");
    printf(" x = %10.3lf\n", x);

    p = &y;
    printf("\np = %p\n", p);
    printf("*p = %10.3lf\n", *p);
    printf(" y = %10.3lf\n", y);
    *p = 100.123;
    printf("After *p = 100.123; ==> \n");
    printf(" y = %10.3lf\n", y);
    return 0;
}
```

```
p = 002AF668
*p = 10.000
x = 10.000
After *p = 50.123; ==>
x = 50.123

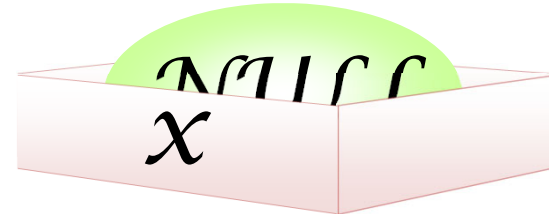
p = 002AF658
*p = 20.000
y = 20.000
After *p = 100.123; ==>
y = 100.123
```



포인터 사용시 주의점

- ◆ 포인터가 아무것도 가리키고 있지 않는 경우에는 **NULL**로 초기화
- ◆ NULL 포인터를 가지고 간접 참조하면 하드웨어로 감지할 수 있다.
- ◆ 포인터의 유효성 여부 판단이 쉽다.

포인터가 아무것도
가리키지 않을때는
반드시 NULL로
설정하세요.



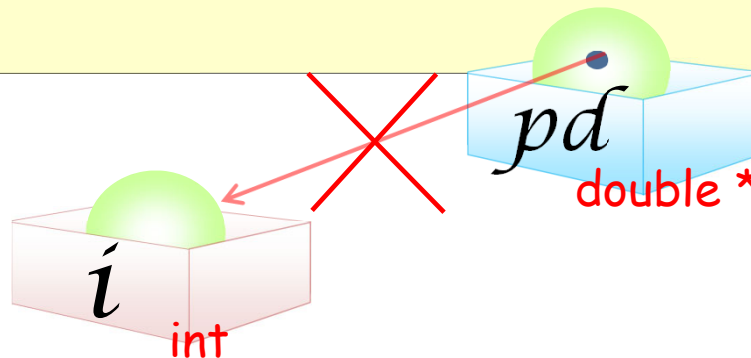
포인터 사용시 주의점

◆ 포인터의 자료형과 변수의 자료형은 일치하여야 한다.

```
#include <stdio.h>
int main(void)
{
    int i;
    → double *pd = NULL;
    double d;

    pd = &i;
    pd = &d;
    *pd = 36.5;
    return 0;
}
```


// 오류! double형 포인터에 int형 변수의 주소를 대입



**포인터 연산,
포인터 (Pointer)와 배열**

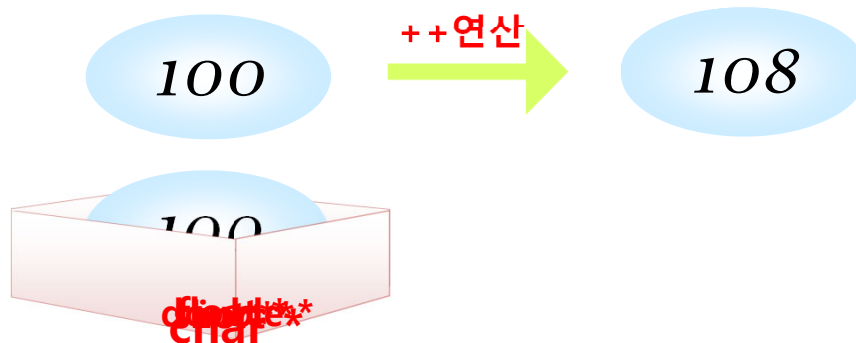
포인터 연산

- ◆ 가능한 연산: 증가, 감소, 덧셈, 뺄셈 연산
- ◆ 증가 연산의 경우 증가되는 값은 포인터가 가리키는 객체 자료형의 크기



포인터 타입	++연산후 증가되는값
Char *	1
Short *	2
int *	4
float *	4
double *	8

포인터의 증가는
일반 변수와는 약간 다르며,
가리키는 객체 자료형의
크기만큼 씩 증가합니다.



포인터의 증감 연산

void test_pointer_arithmetics()

```
{
    char *pc = (char *)0x1000;
    int *pi = (int *)0x1000;
    double *pd = (double *)0x1000;

    printf("Incrementing pointers: \n");
    for (int i = 0; i < 5; i++)
    {
        printf("%d: pc (%p), pi(%p), pd(%p)\n", i, pc, pi, pd);
        pc = pc + 1;
        pi = pi + 1;
        pd = pd + 1;
    }

    printf("Decrementing pointers: \n");
    for (int i = 0; i < 5; i++)
    {
        pc--;
        pi--;
        pd--;
        printf("%d: pc (%p), pi(%p), pd(%p)\n", i, pc, pi, pd);
    }
}
```

```
Incrementing pointers:
0: pc (00001000), pi(00001000), pd(00001000)
1: pc (00001001), pi(00001004), pd(00001008)
2: pc (00001002), pi(00001008), pd(00001010)
3: pc (00001003), pi(0000100C), pd(00001018)
4: pc (00001004), pi(00001010), pd(00001020)
Decrementing pointers:
0: pc (00001004), pi(00001010), pd(00001020)
1: pc (00001003), pi(0000100C), pd(00001018)
2: pc (00001002), pi(00001008), pd(00001010)
3: pc (00001001), pi(00001004), pd(00001008)
4: pc (00001000), pi(00001000), pd(00001000)
계속하려면 아무 키나 누르십시오 . . .
```

◆ 포인터의 증감 연산 결과분석

- char pointer → 1씩 증감
- int pointer → 4씩 증감
- double pointer → 8씩 증감

➔ 포인터의 증감연산에서는
그 포인터의 데이터 유형의
크기에 따라 증감됨



간접 참조 연산자와 증감 연산자를 함께 사용하는 경우

◆ ***p++;**

- p가 가리키는 위치에서 값을 가져온 후에 p를 증가한다.
- *p와 p++를 분리하여 실행 가능

◆ **(*p)++;**

- p가 가리키는 위치의 값을 증가한다.

수식	의미
$v = *p++$	p가 가리키는 값을 v에 대입한 후에 p를 증가한다. ($v = *p; p++;$)
$v = (*p)++$	p가 가리키는 값을 v에 대입한 후에 가리키는 값을 증가한다. ($v = *p; *p = *p + 1;$)
$v = *++p$	p를 증가시킨 후에 p가 가리키는 값을 v에 대입한다. ($++p; v = *p;$)
$v = ++*p$	p가 가리키는 값을 가져온 후에 그 값을 증가하여 v에 대입한다. ($x = *p; v = x + 1;$)



포인터의 형 변환 (pointer type conversion)

◆ C언어에서는 꼭 필요한 경우에, 명시적으로 포인터의 타입을 변경할 수 있다.

◆ **calloc**의 예

- void * **calloc**(size_t num_elements, size_t size_of_element);
- void * 형은 다른 포인터 형으로 지정하여 사용하라는 의미
- 동적 메모리 할당 예)

```
double *array = NULL;
int size = 100;

array = (double *)calloc(size, sizeof(double));
```



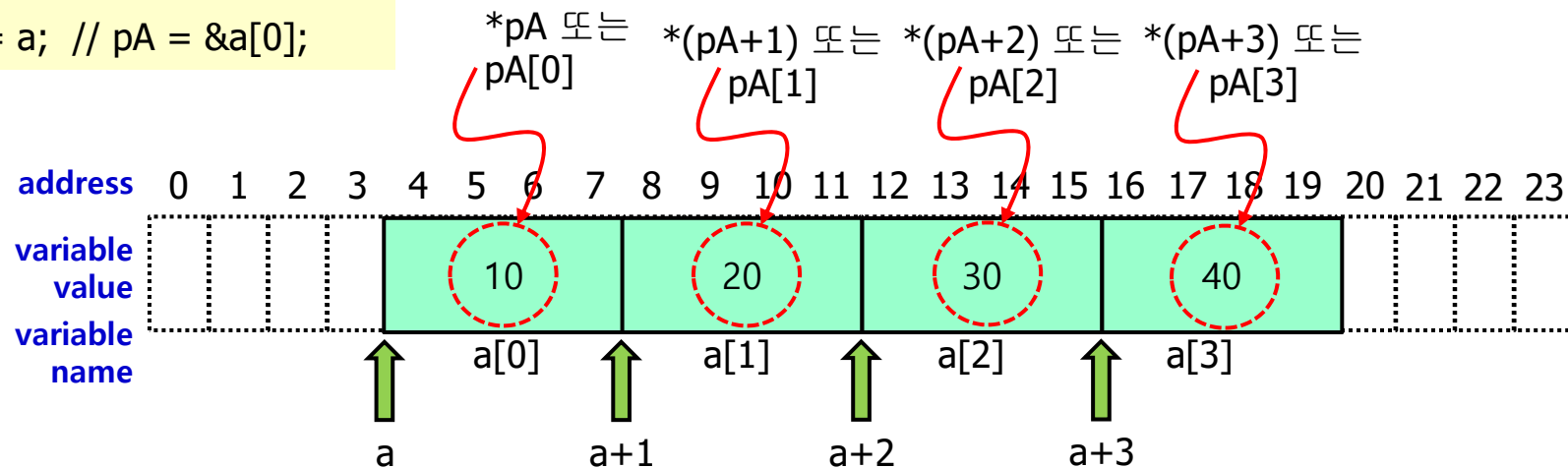
포인터와 배열

- ◆ 배열과 포인터는 아주 밀접한 관계를 가지고 있다.
- ◆ 배열 이름은 주소값을 변경할 수 없는 포인터이다.
- ◆ 포인터는 배열 이름처럼 사용이 가능하다.
- ◆ 즉, 배열의 인덱스 표기법을 포인터에 사용하여 배열처럼 사용할 수 있다.

```
int a[4] = {10, 20, 30, 40};
```

```
int *pA = NULL;
```

```
pA = a; // pA = &a[0];
```



포인터를 배열처럼 사용

```
void test_pointer_and_array()
{
    int a[] = { 10, 20, 30, 40 };
    int *p = NULL;

    p = a;
    printf("a[0] = %2d a[1] = %2d a[2] = %2d a[3] = %2d\n", a[0], a[1], a[2], a[3]);
    printf("p[0] = %2d p[1] = %2d p[2] = %2d p[3] = %2d\n", p[0], p[1], p[2], p[3]);
    printf("\n");

    p[0] = 60; p[1] = 70; p[2] = 80; p[3] = 90;

    printf("a[0] = %2d a[1] = %2d a[2] = %2d a[3] = %2d\n", a[0], a[1], a[2], a[3]);
    printf("p[0] = %2d p[1] = %2d p[2] = %2d p[3] = %2d\n", p[0], p[1], p[2], p[3]);
}
```

```
a[0] = 10 a[1] = 20 a[2] = 30 a[3] = 40
p[0] = 10 p[1] = 20 p[2] = 30 p[3] = 40

a[0] = 60 a[1] = 70 a[2] = 80 a[3] = 90
p[0] = 60 p[1] = 70 p[2] = 80 p[3] = 90
계속하려면 아무 키나 누르십시오 . . .
```



포인터를 사용한 방법의 장점

◆ 포인터가 인덱스 표기법보다 빠르다.

- Why?: 원소의 주소를 계산할 필요가 없다.

```
int get_sum1(int a[], int n)
{
    int i;
    int sum = 0;

    for(i = 0; i < n; i++ )
    {
        sum += a[i];
    }
    return sum;
}
```

인덱스 표기법 사용

```
int get_sum2(int a[], int n)
{
    int i, sum = 0;
    int *p = NULL;
    p = a;
    for(i = 0; i < n; i++ )
    {
        sum += *p++;
    }
    return sum;
}
```

포인터 사용



함수 호출에서의 포인터 (Pointer) 인수
- Call-by-pointer, return-by-pointer

함수의 인수 (argument) 전달 방법

◆ C에서 함수의 인수 전달 방법

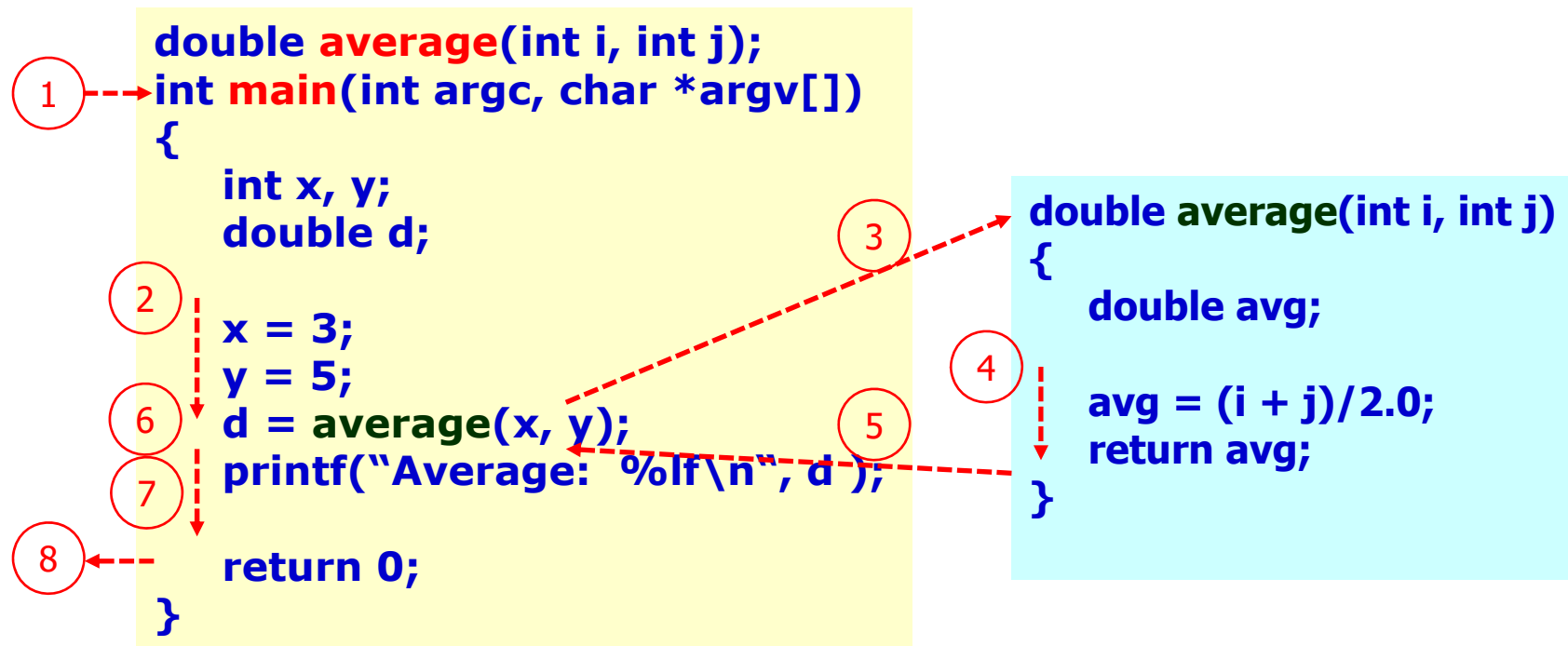
- 값에 의한 호출 (call-by-value): 기본적인 방법
- 포인터에 의한 호출 (call-by-pointer): 포인터 이용
- 참조에 의한 호출 (call-by-reference): 참조 (reference) 이용



값에 의한 호출 (call-by-value)

◆ 함수 호출시에 변수의 값을 함수에 복사본으로 전달

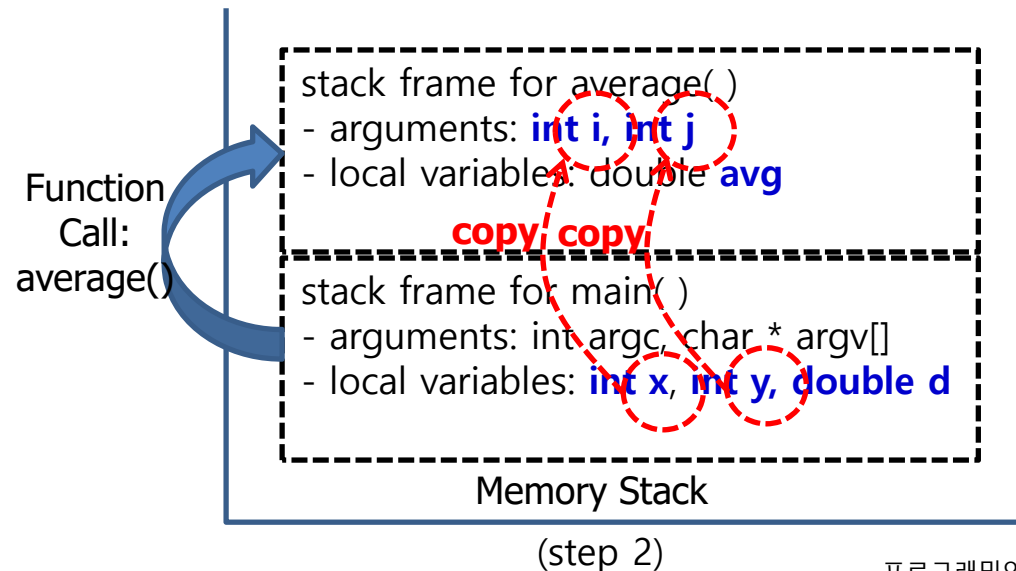
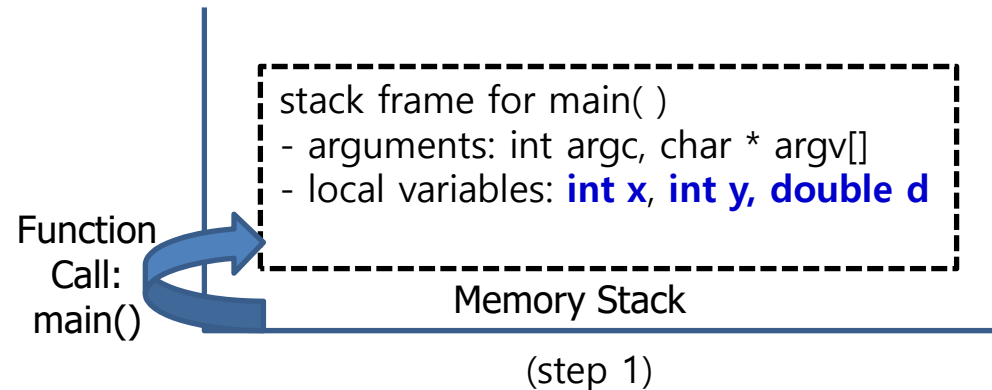
- 복사본이 전달되며, 호출하는 (calling) 함수의 변수와 호출된 (called) 함수의 인수 (argument)는 별도의 변수로 관리됨
- 호출된 함수의 인수가 변경되어도, 호출한 함수의 변수는 변경없음



Argument passing in Call-by-Value

```
double average(int i, int j);  
void main(int argc, char *argv[])  
{  
    int x, y;  
    double d;  
  
    x = 3;  
    y = 5;  
    d = average(x, y);  
    printf("Average: %lf\n", d );  
}
```

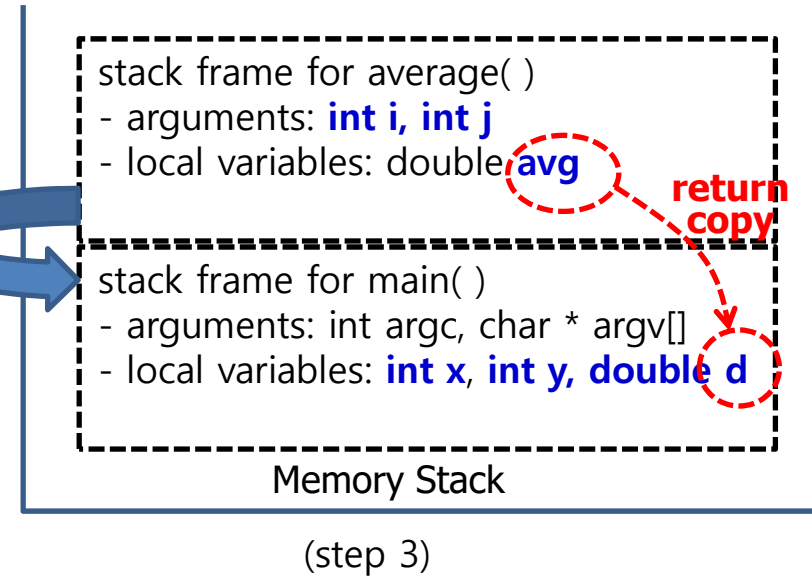
```
double average(int i, int j)  
{  
    double avg;  
  
    avg = (i + j)/2.0;  
    return avg;  
}
```



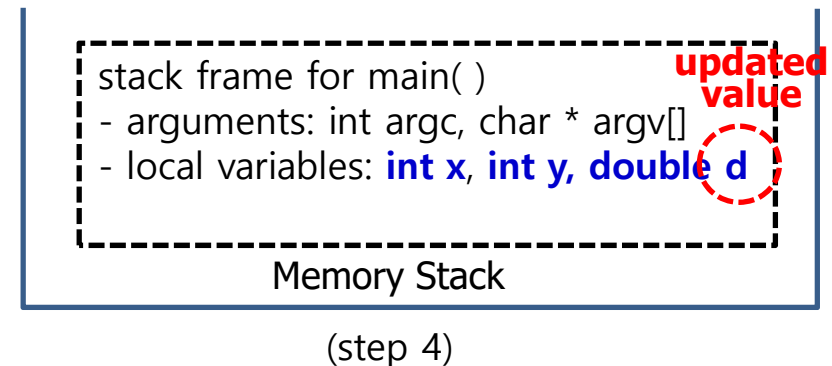
Argument passing in Call-by-Value

```
double average(int i, int j);  
void main(int argc, char *argv[])  
{  
    int x, y;  
    double d;  
  
    x = 3;  
    y = 5;  
    d = average(x, y);  
    printf("Average: %lf\n", d );  
}
```

Function
Return



```
double average(int i, int j)  
{  
    double avg;  
  
    avg = (i + j)/2.0;  
    return avg;  
}
```



포인터에 의한 호출 (call-by-pointer)

◆ 함수 호출 시에 변수의 주소 정보 (포인터)를 함수의 인수로 전달

```
void main()
{
    double d = 123.456;
    double result;

    . . . .

    result = function(&d);

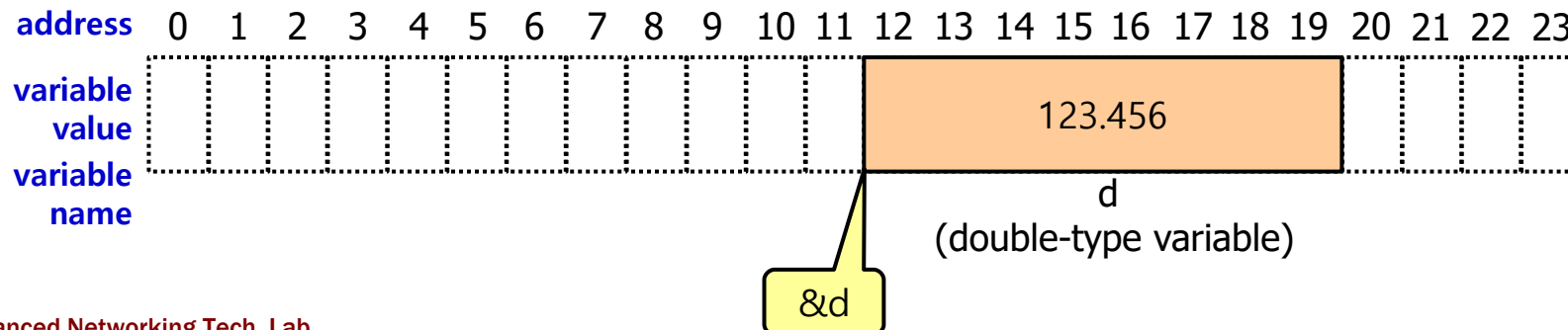
    . . . .
}
```

Call-by-Pointer에서는
변수의 주소 정보를
인수로 전달

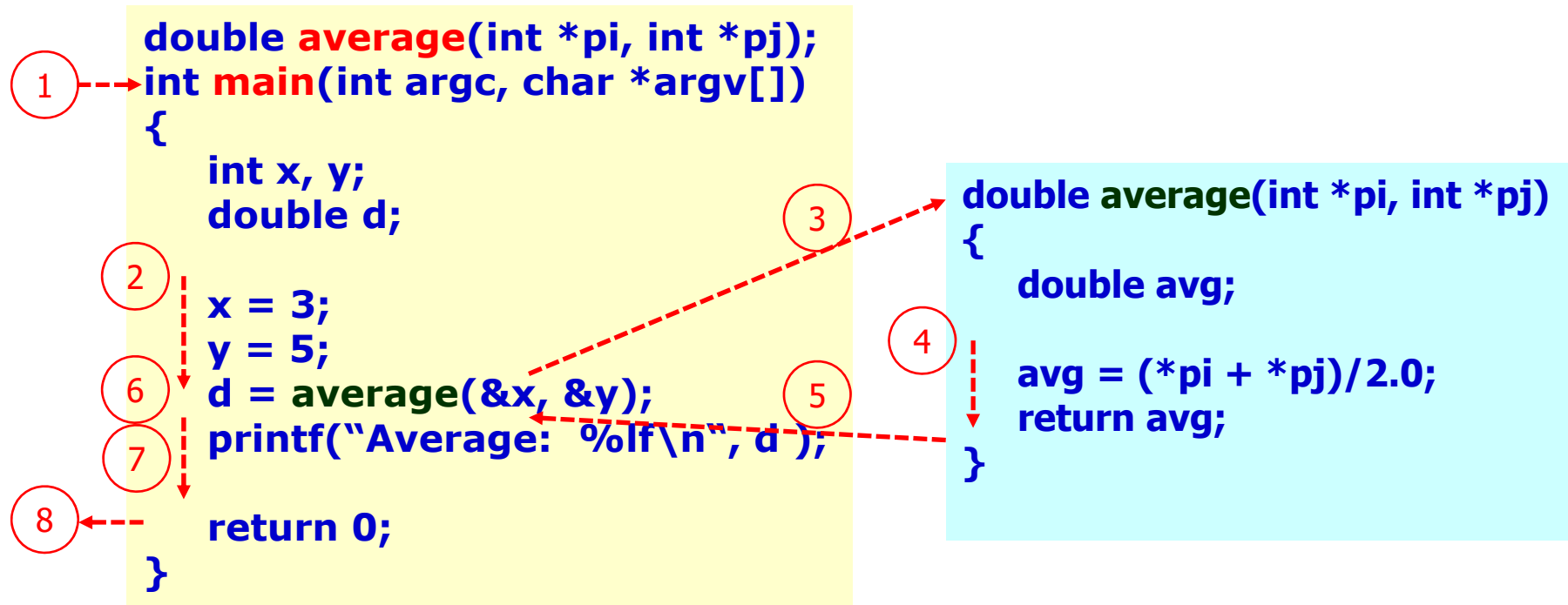
```
double function(double *pD)
{
    double res;

    res = (*pD) * (*pD);

    return res;
}
```



Argument passing in Call-by-Pointer



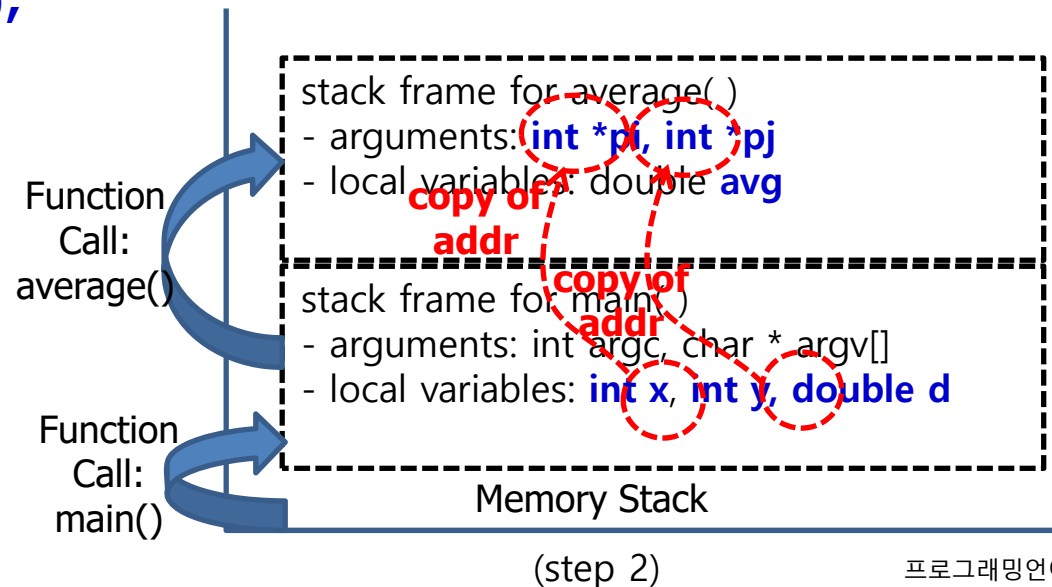
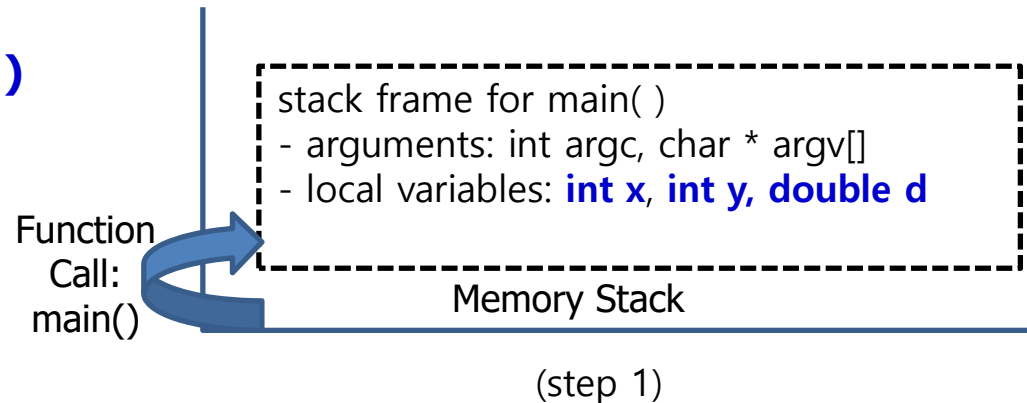
Argument passing in Call-by-Pointer

```
double average(int *pi, int *pj);
void main(int argc, char *argv[])
{
    int x, y;
    double d;

    x = 3;
    y = 5;
    d = average(&x, &y);
    printf("Average: %lf\n", d );
}
```

```
double average(int *pi, int *pj)
{
    double avg;

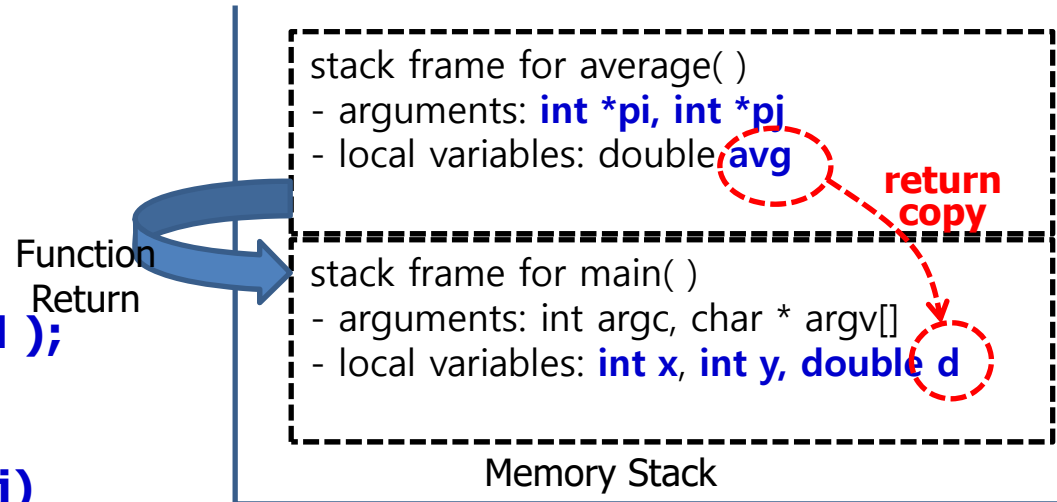
    avg = (*pi + *pj)/2.0;
    return avg;
}
```



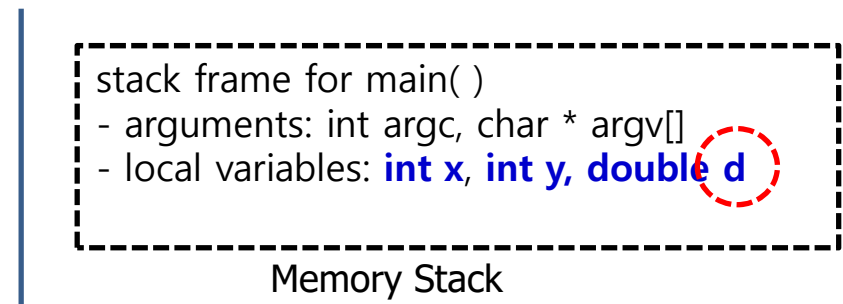
Argument passing in Call-by-Pointer

```
double average(int *pi, int *pj);  
void main(int argc, char *argv[])  
{  
    int x, y;  
    double d;  
  
    x = 3;  
    y = 5;  
    d = average(&x, &y);  
    printf("Average: %lf\n", d );  
}
```

```
double average(int *pi, int *pj)  
{  
    double avg;  
  
    avg = (*pi + *pj)/2.0;  
    return avg;  
}
```



(step 3)



(step 4)



참조에 의한 호출 (call-by-reference)

◆ 함수 호출 시에 변수의 주소 정보 (참조)를 함수의 매개 변수로 전달

- 함수의 호출 시에 calling 함수에서의 호출은 call-by-value에서와 동일
- 호출된 함수에서의 parameter에는 & 표시가 데이터 유형 다음에 표시됨
- call-by-reference의 경우, 호출된 함수에서 호출한 함수의 변수 값을 변경할 수 있음

◆ 예)

```
double average(int i, int j); // call-by-value
double average(int& i, int& j); // call-by-reference
void main(int argc, char *argv[])
{
    int x, y;
    double d;

    ....
    d = average(x, y);
    ....
}
```



Argument passing in Call-by-Reference

```
void average(int i, int j, int& sum, double& avg);
```

```
void main(int argc, char *argv[])
```

```
{
```

```
    int x, y, sum;  
    double avg;
```

```
    x = 3;
```

```
    y = 5;
```

```
    average(x, y, sum, avg);
```

```
    printf("Sum: %d,  
          Average: %lf\n", sum, avg);
```

```
}
```

```
void average(int i, int j,  
            int& sum, double& avg)
```

```
{
```

```
    sum = i + j;
```

```
    avg = sum/2.0;
```

```
}
```

Function
Call:
main()

stack frame for main()
- arguments: int argc, char * argv[]
- local variables: **int x, y, sum, double avg**

Memory Stack

(step 1)

Function
Return

Function
Call:
average()

stack frame for average()
- arguments: **int i, j, int& sum, double& avg**
- local variables:

stack frame for main()
- arguments: int argc, char * argv[]
- local variables: **int x, y, sum, double avg**

Memory Stack

(step 2, 3)



함수의 인수 전달 방식의 비교

```
/* TestArgumentPassing.c */
#include <stdio.h>
void swap_call_by_value (int x, int y);
void swap_call_by_pointer(int *px, int *py);
void swap_call_by_reference(int &x, int &y);

void main()
{
    int x, y;

    x = 10; y = 30;
    printf("\nBefore call-by-value: x = %d, y = %d\n", x, y);
    swap_call_by_value(x, y);
    printf("After call-by-value: x = %d, y = %d\n", x, y);

    x = 5; y = 7;
    printf("\nBefore call-by-pointer: x = %d, y = %d\n", x, y);
    swap_call_by_pointer(&x, &y);
    printf("After call-by-pointer: x = %d, y = %d\n", x, y);

    x = 25; y = 50;
    printf("\nBefore call-by-reference: x = %d, y = %d\n",
        x, y);
    swap_call_by_reference(x, y);
    printf("After call-by-reference: x = %d, y = %d\n", x, y);
    printf("\n");
}
```

```
void swap_call_by_value(int a, int b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
}

void swap_call_by_pointer(int *pa, int *pb)
{
    int temp;
    temp = *pa;
    *pa = *pb;
    *pb = temp;
}

void swap_call_by_reference(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

```
Before call-by-value: x = 10, y = 30
After call-by-value: x = 10, y = 30
```

```
Before call-by-pointer: x = 5, y = 7
After call-by-pointer: x = 7, y = 5
```

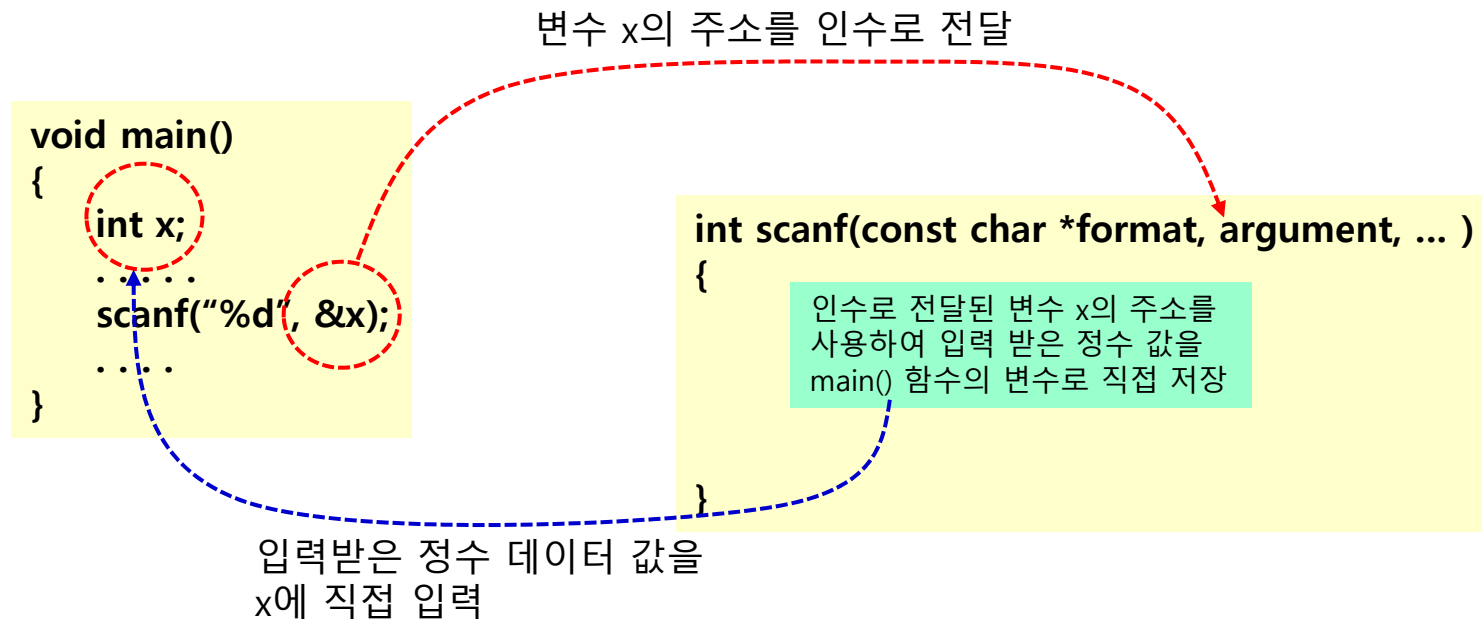
```
Before call-by-reference: x = 25, y = 50
After call-by-reference: x = 50, y = 25
```



scanf() 함수

◆ scanf() 함수

- 변수에 값을 직접 저장하기 위하여 함수 호출에서 인수로 변수의 주소를 전달받는다.




2개 이상의 결과를 반환하는 방법

```
#include <stdio.h>
// 기울기와 y절편을 계산
int get_line_parameter(int x1, int y1, int x2, int y2,
    double *slope, double *yintercept)
{
    if( x1 == x2 )
        return -1;
    else {
        *slope = (double)(y2 - y1)/((double)(x2 - x1));
        *yintercept = y1 - (*slope)*x1;
        return 0;
    }
}

int main(void)
{
    double s, y;
    if( get_line_parameter(3, 3, 6, 6, &s, &y) == -1 )
        printf("에러\n");
    else
        printf("기울기는 %lf\n, y절편은 %lf\n", s, y);
    return 0;
}
```

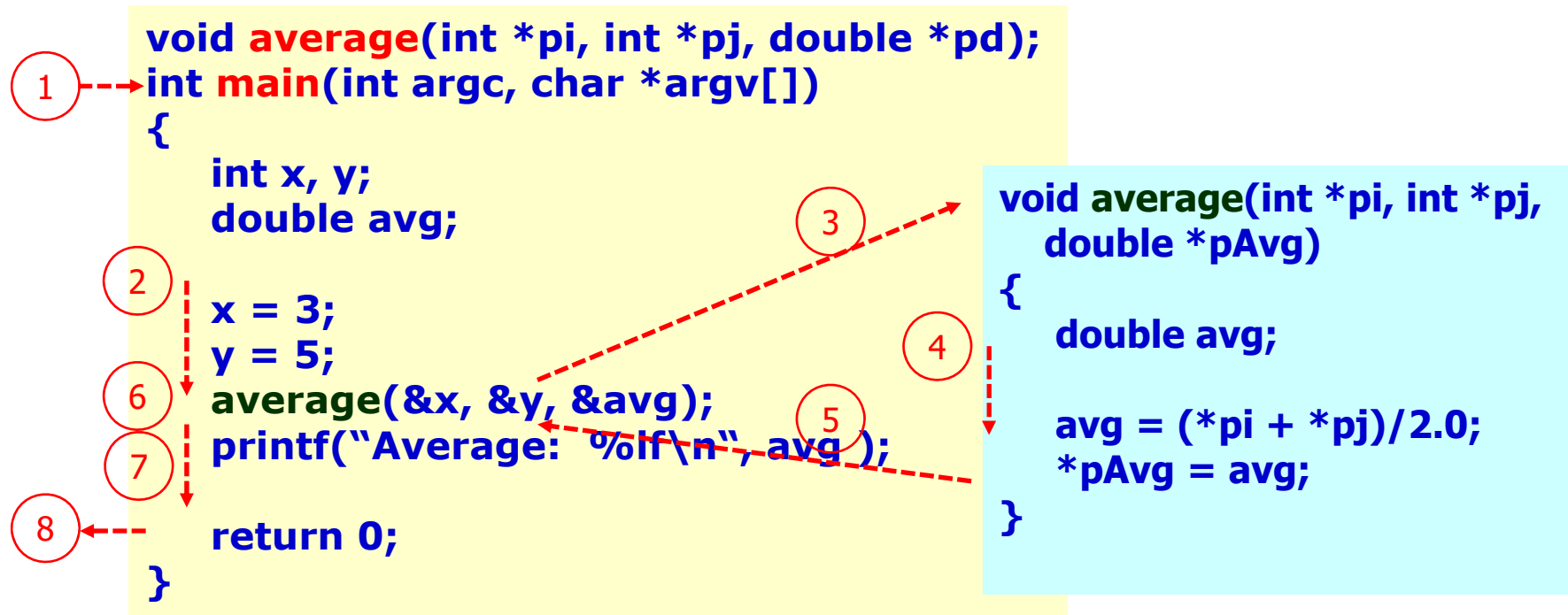
기울기와 y절편을 인수로 전달



기울기는 1.000000,
y절편은 0.000000



Argument passing in Call-by-Pointer, Return-by-Pointer



배열을 함수의 인수로 전달하는 경우

◆ 일반 변수 vs 배열

```
// 매개 변수 x에 기억 장소가 할당  
void sub(int x)  
{  
    ...  
}
```

```
// b[]에 매개변수 기억 장소가 할당되지 않는다.  
void sub(int b[], int n)  
{  
    ...  
}
```

◆ 배열의 경우, 크기가 큰 경우에 복사하려면 많은 시간 소모

◆ 배열의 경우, 배열의 주소를 전달

배열의 원소들을 모두 복사하여 전달하지 않고, 배열의 주소만 전달

```
double sub_func(int array[], int size);  
  
void main()  
{  
    int data[10000]; // 지역 변수  
    ...  
    sub_func(data, 10000);  
}
```

```
double sub_func(int array[], int size)  
{  
    double result;  
    ...  
    return result;  
}
```



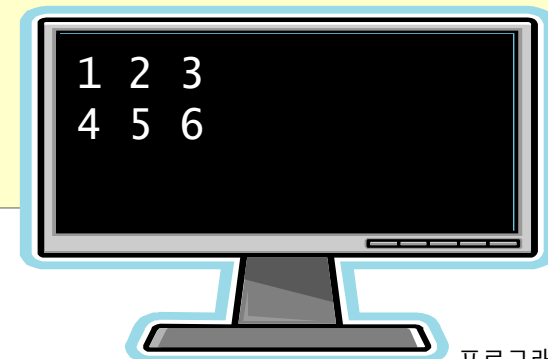
예제

```
#include <stdio.h>
void sub(int b[], int n);
```

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

    printf("%d %d %d\n", a[0], a[1], a[2]);
    sub(a, 3);
    printf("%d %d %d\n", a[0], a[1], a[2]);
    return 0;
}

void sub(int b[], int n)
{
    b[0] = 4;
    b[1] = 5;
    b[2] = 6;
}
```



포인터를 사용하여 결과값을 반환할 때 주의점

- ◆ 함수가 종료되더라도 남아 있는 변수의 주소를 반환하여야 한다.
- ◆ 지역 변수의 주소를 반환하면 , 함수가 종료되면 사라지기 때문에 오류

```
int *add(int x, int y)
{
    int result;

    result = x + y;
    return &result;
}
```

지역변수 result는
함수가 종료되면 소멸되므로 그
주소를 반환할 수 없다!

```
void add(int x, int y, int *sum)
{
    int result;

    result = x + y;
    *sum = result;
}
```

함수 호출에서 결과값을 저장할
주소를 포인터인수로 전달하여
직접 저장



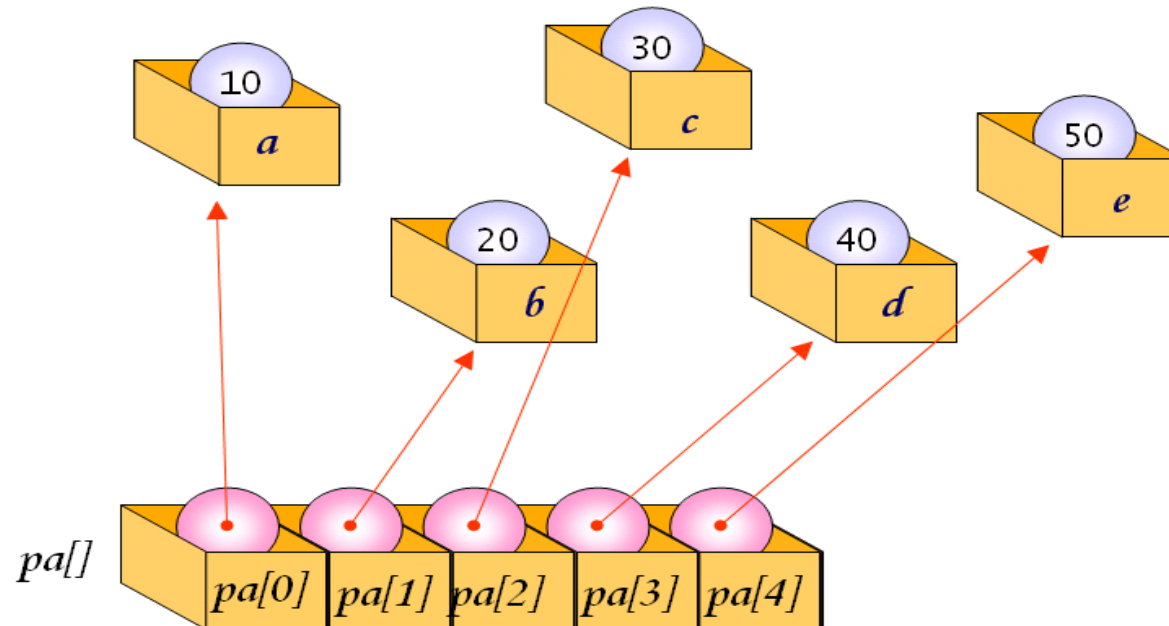
포인터 (Pointer) 배열

포인터 배열 (array of pointers)

◆ 포인터 배열(array of pointers)

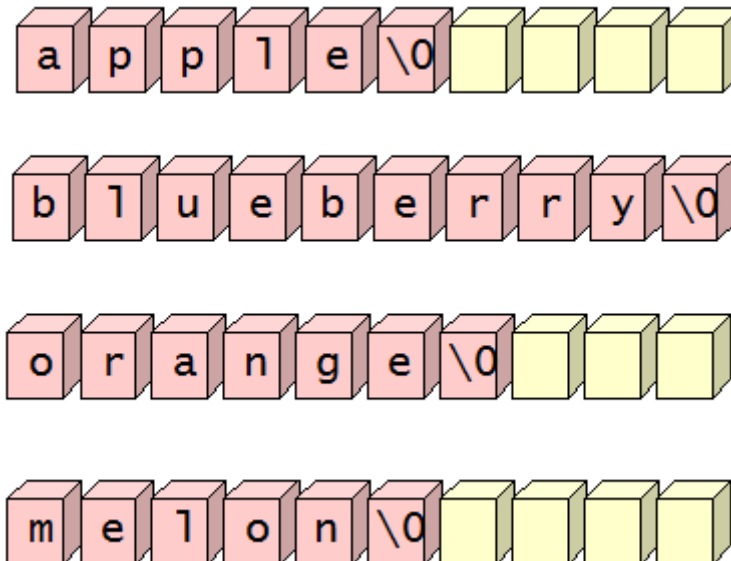
- 포인터를 모아서 배열로 만든 것
- 배열의 원소들이 포인터

```
int a = 10, b = 20, c = 30, d = 40, e = 50;  
int *pa[5] = { &a, &b, &c, &d, &e };
```



2차원 문자 (character) 배열에 문자열 (string)을 저장

```
char fruits[4 ][10] = {  
    "apple",  
    "blueberry",  
    "orange",  
    "melon"  
};
```



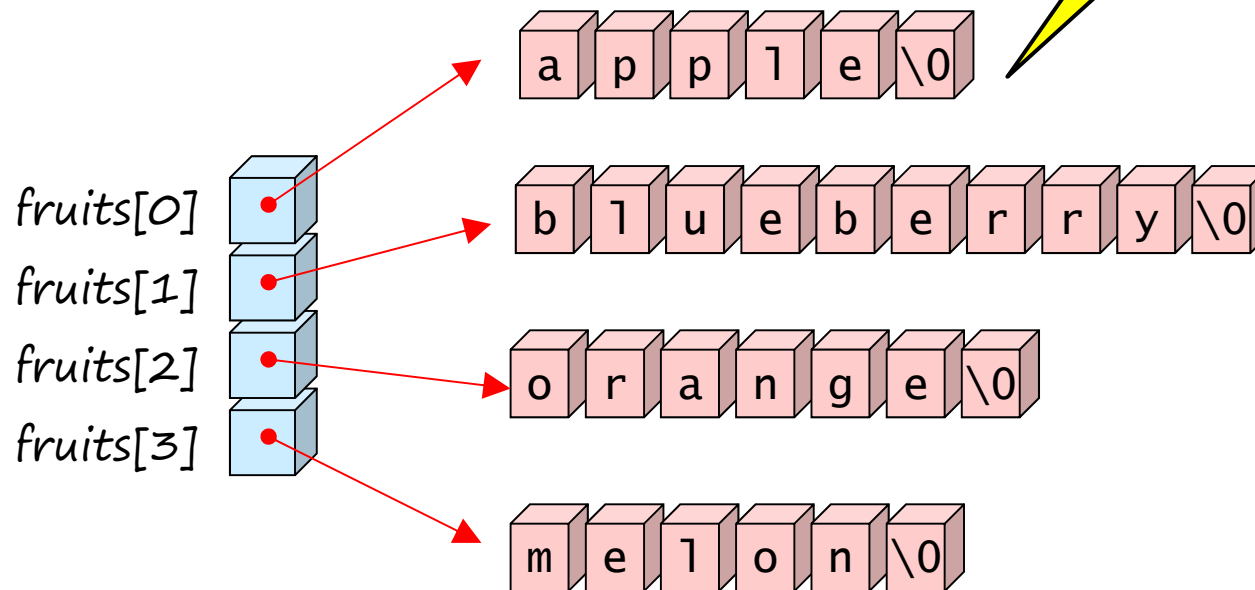
문자열 (string) 저장에 문자형 2차원 배열을 사용하는 경우 낭비되는 공간이 발생합니다 !



문자형 포인터 배열

```
const char *fruits[ ] = {  
    "apple",  
    "blueberry",  
    "orange",  
    "melon"  
};
```

문자열 (string) 저장에 각
문자열의 문자형 포인터들
을 배열로 구성하면 낭비
되는 공간이 발생되지 않
습니다 !



예제

// 문자열 배열

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

```
int main(void)
```

```
{
```

```
    int i, n;
```

```
    const char *fruits[ ] = {  
        "apple",  
        "blueberry",  
        "orange",  
        "melon"  
    };
```

```
    // 배열 원소 개수 계산
```

```
    n = sizeof(fruits)/sizeof(fruits[0]);
```

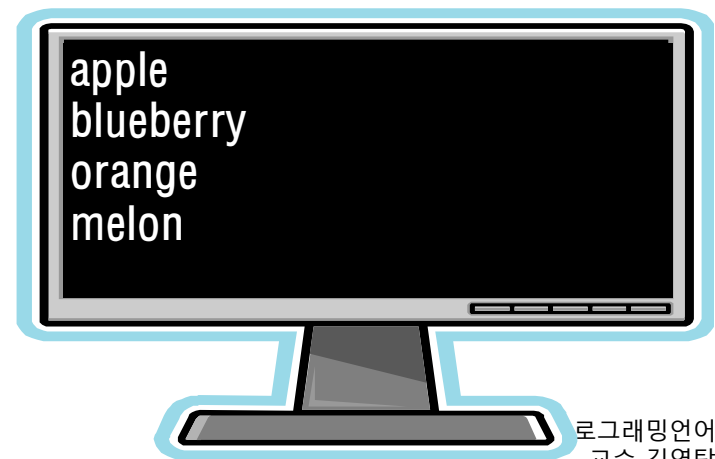
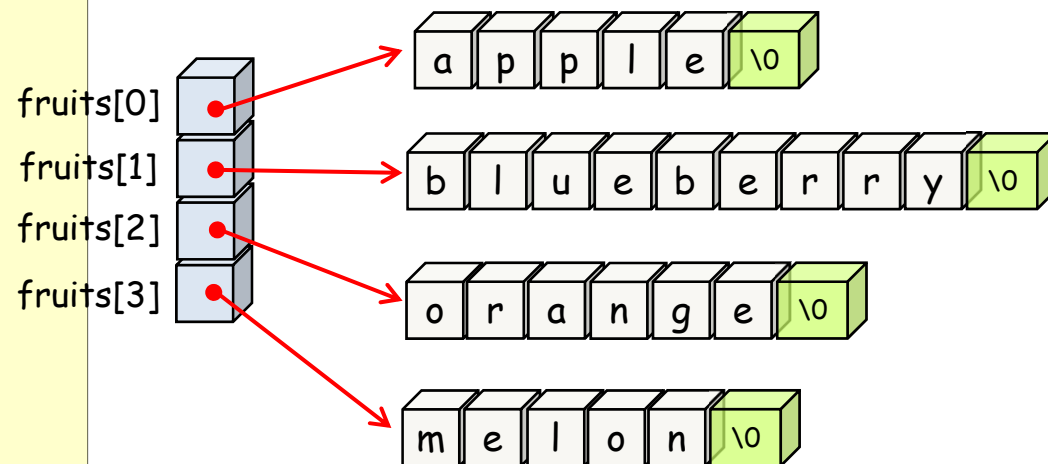
```
    for(i = 0; i < n; i++)
```

```
        printf("%s \n", fruits[i]);
```

```
    return 0;
```

```
}
```

각각의 문자열의 길이가 달라도 메모리의 낭비가 발생하지 않는다.



동적 메모리 할당 (Dynamic Memory Allocation)
동적 배열 (Dynamic Array)

동적 메모리 할당의 개념

◆ 프로그램이 메모리를 할당 받는 방법

● auto 지역 변수의 메모리 할당

- 프로그램 소스코드에 배열을 선언:
`#define ARRAY_SIZE 100`
`int data_array[ARRAY_SIZE];`
- 프로그램 실행 단계에서 필요한 크기가 변경되는 것에 상관없이 항상 일정한 크기를 유지
- 항상 예상되는 최대 크기의 배열을 선언해야 하며, 메모리 낭비가 발생할 수 있음

● 동적 할당(dynamic allocation)

- 프로그램 실행 단계에서 필요에 따라 동적으로 배열을 생성:
`int array_size;`
`int *pArray;`
`scanf("%d", &array_size);`
`pArray = (int *)calloc(array_size, sizeof(int));`
- 프로그램 실행 단계에서 필요한 크기가 변경된 것 만큼의 메모리 할당
- 항상 필요한 크기의 배열을 구성할 수 있어 메모리 사용에 낭비가 없음



자동 (auto) 지역 변수의 메모리 할당

◆ 자동 (auto) 지역 변수 (local variable)의 메모리 할당

- 프로그램이 시작되기 전에 미리 정해진 크기의 메모리를 할당 받는 것
- 메모리의 크기는 프로그램이 시작하기 전에 결정
- 처음에 결정된 크기보다 더 큰 입력이 들어온다면 처리하지 못함
- 더 작은 입력이 들어온다면 남은 메모리 공간은 낭비

```
#define MAX_NUM_STUDENTS 100  
  
int score_s[MAX_NUM_STUDENTS];  
  
score_s[10] = 99.5;
```

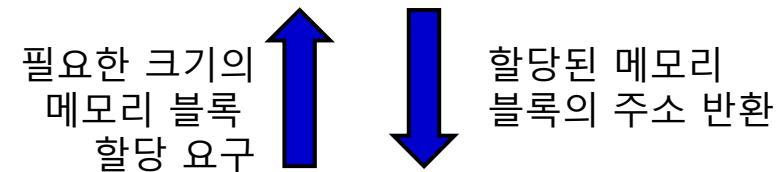


동적 (dynamic) 메모리 할당

◆ 동적 메모리 할당

- 실행 도중에 동적으로 메모리를 할당 받는 것
- 사용이 끝나면 시스템에 메모리를 반납
- 필요한 만큼만 할당을 받고 메모리를 매우 효율적으로 사용
- `calloc()`, `malloc()` 계열의 라이브러리 함수를 사용

운영체제 (Operating System)
동적 메모리 할당 관리



```
int array_size;  
int *intArray;  
  
printf("Input array_size : ");  
scanf("%d", &array_size);  
intArray = (int *)calloc(array_size, sizeof(int));  
if( intArray == NULL )  
{  
    ... // 오류 처리  
}  
intArray[5] = 30;  
  
free(intArray);
```



동적 메모리 블록 할당 및 반환 함수

분류	함수 원형과 인수	기능
동적 메모리 블록 할당 및 반환 <stdlib.h>	<code>void* malloc(size_t size)</code>	지정된 size 크기의 메모리 블록을 할당하고, 그 시작 주소를 void pointer로 반환
	<code>void *calloc(size_t num_element, size_t element_size)</code>	element_size 크기의 항목을 num_element개 할당하고, 0으로 초기화 한 후, 그 시작 주소를 void pointer로 반환
	<code>void *realloc(void *p, size_t size)</code>	이전에 할당받아 사용하고 있는 메모리 블록의 크기를 변경 p는 현재 사용하고 있는 메모리 블록의 주소, size는 변경하고자 하는 크기; 기존의 데이터 값은 유지된다
	<code>void free(void *p)</code>	동적 메모리 블록을 시스템에 반환, p는 현재 사용하였던 메모리 블록 주소



Heap Memory Management

◆ Heap

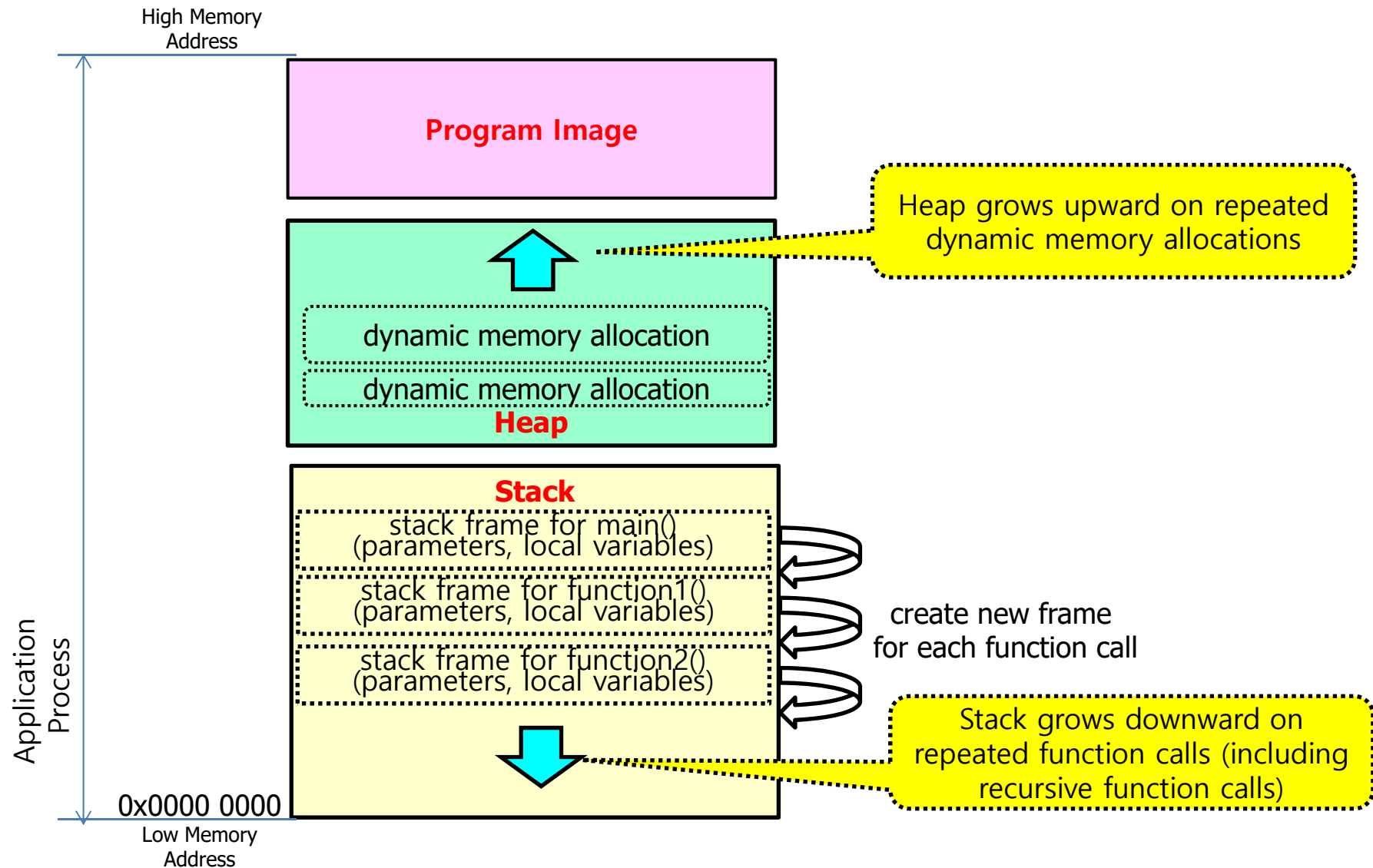
- Also called "freestore"
- Reserved for dynamically-allocated variables
- All new dynamic variables consume memory in freestore
 - If too many → could use all freestore memory

◆ Heap에 남아 있는 메모리가 없는 경우

- calloc() 또는 malloc() 함수를 통한 메모리 블록 할당 요청에 대하여 추가적인 메모리 블록 할당을 할 수 없게 되며, NULL 값을 반환

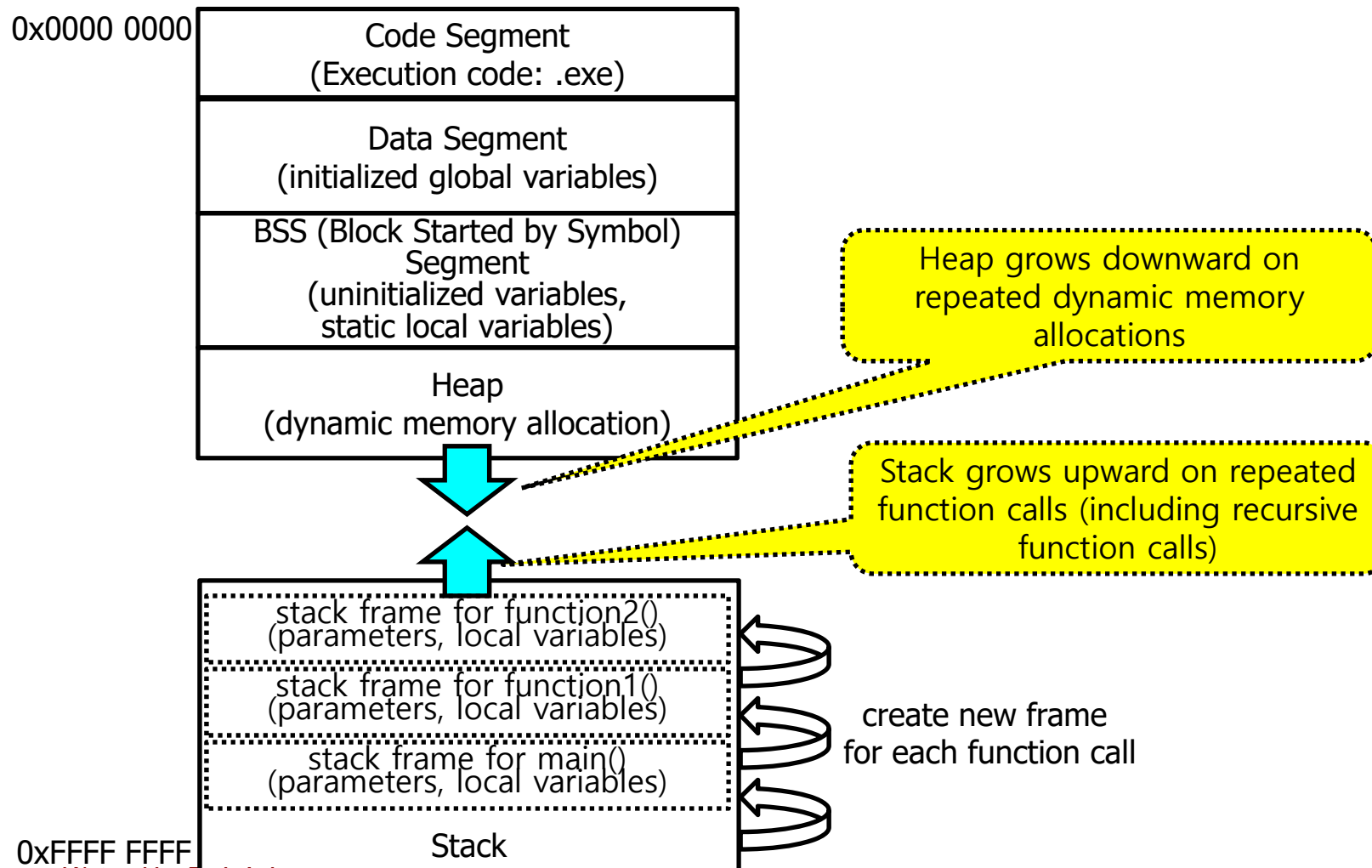


Memory Map (MS-Windows)



Memory Map

◆ Memory Map of a process (UNIX/Linux O.S.)



동적 메모리 할당

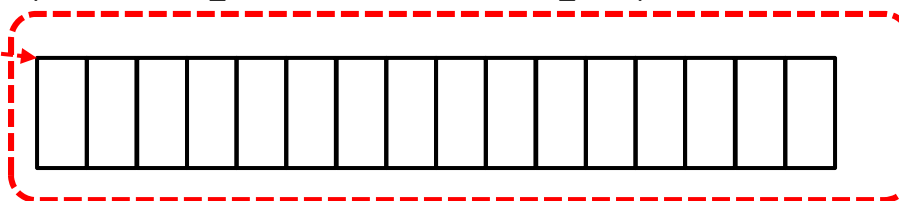
◆ void ***calloc**(size_t num_elements, size_t element_size)

- size는 바이트의 수
- calloc()함수는 메모리 블록의 첫 번째 바이트에 대한 주소를 반환
- 만약 요청한 메모리 공간을 할당할 수 없는 경우에는 NULL값을 반환

```
int array_size;  
int *intArray = NULL;  
scanf("%d", &array_size);  
intArray = (int *)calloc(array_size, sizeof(int));  
if( intArray == NULL )  
{  
    ... // 오류 처리  
}
```

동적으로 할당된 메모리 블록
(크기: num_elements x element_size)

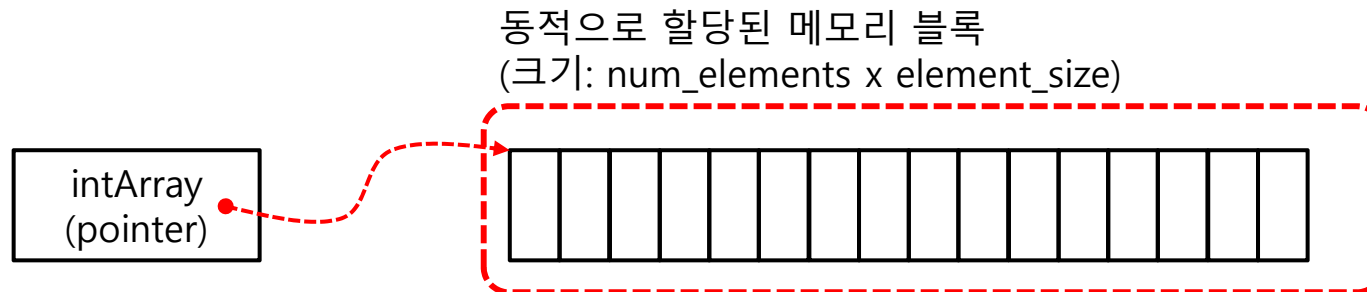
intArray
(pointer)



동적 메모리 할당 블록을 배열로 사용

◆ 동적으로 할당된 메모리 블록을 배열과 같이 사용 가능

- `intArray[0] = 100;`
- `intArray[1] = 200;`
- `intArray[2] = 300;`
- ...



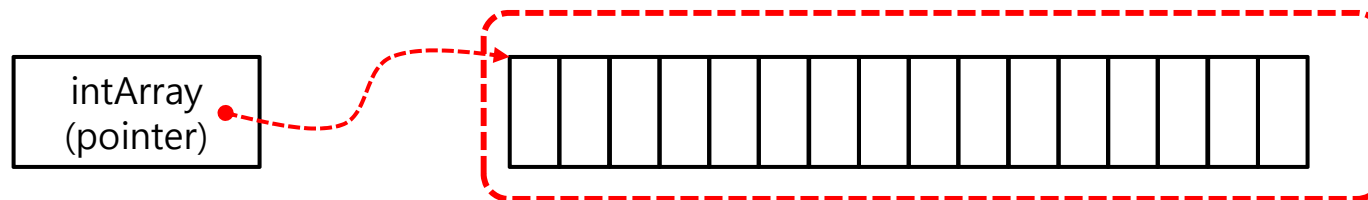
동적 메모리 반납

◆ void free(void *ptr)

- free()는 동적으로 할당되었던 메모리 블록을 시스템에 반납
- ptr은 calloc()을 이용하여 동적 할당된 메모리를 가리키는 포인터

```
int size;  
int *intArray = NULL;  
scanf("%d", &size);  
intArray = (int *)calloc(size, sizeof(int));  
  
...  
free(intArray);  
intArray = NULL; // 메모리 블록 반납 후에는 반드시 NULL로 설정
```

포인터 intArray가 가리키는 메모리 블록을 반납
(크기: num_elements x element_size)



2차원 배열의 동적 생성

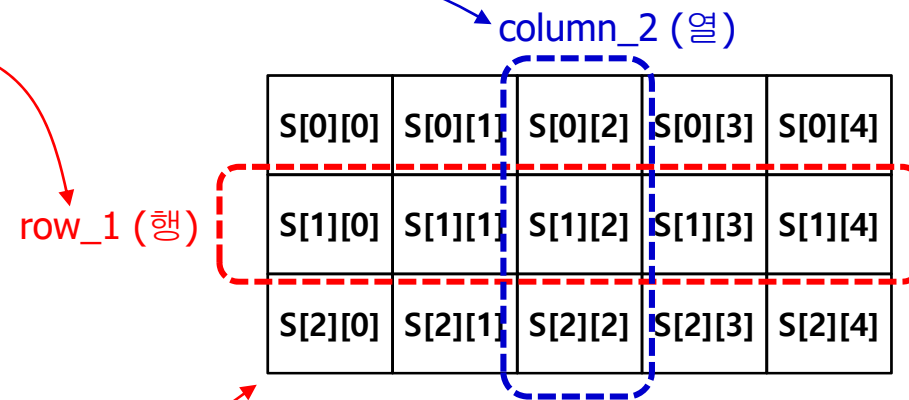
2차원 배열

```
int s_1D[10];    // 1차원 배열  
int s[3][10];    // 2차원 배열  
int s_3D[5][3][10]; // 3차원 배열
```

int s[3][5]; // 과목별 학생성적

첫번째 인덱스: 과목번호

두번째 인덱스: 학생번호(학번)



3 × 5 정수형 2차원 배열



2차원 배열의 활용

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

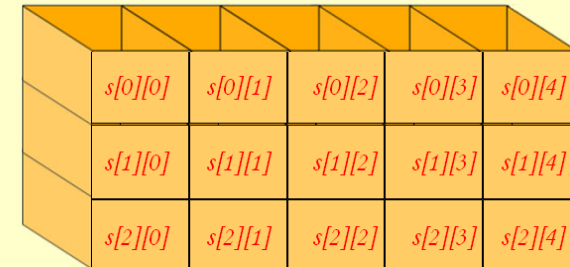
```
int main(void)
```

```
{
```

```
    int s[3][5];        // 2차원 배열 선언  
    int i, j;           // 2개의 인덱스 변수  
    int value = 0;      // 배열 원소에 저장되는 값
```

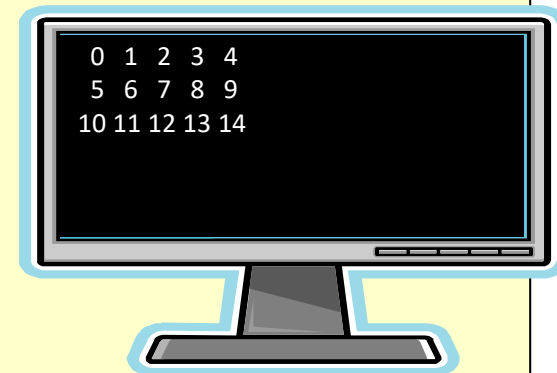
```
    for(i=0; i<3; i++)  
        for(j=0; j<5; j++)  
            s[i][j] = value++;
```

```
    for(i=0; i<3; i++) // row index  
    {  
        for(j=0; j<5; j++) // column index  
            printf("%4d ", s[i][j]);  
        printf("\n");  
    }  
    return 0;  
}
```



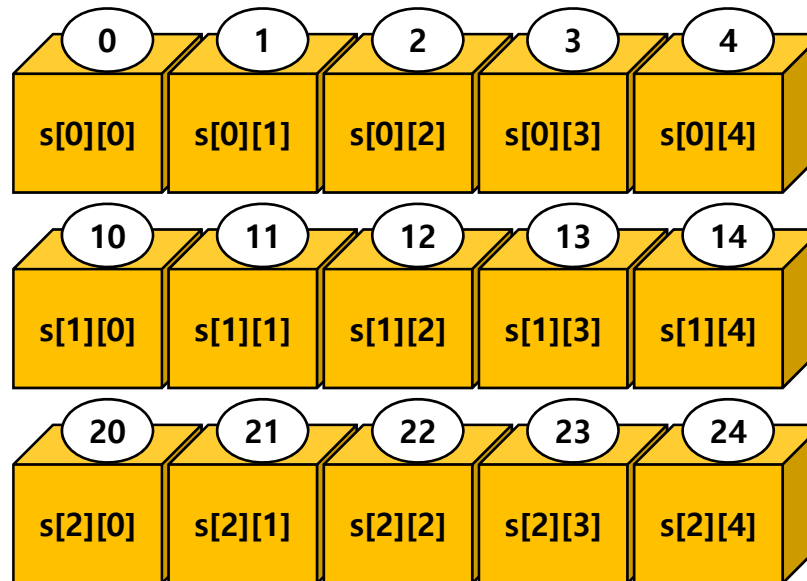
A 3D perspective diagram of a 3x5 array. It consists of three rows and five columns of orange boxes. Each box contains a red text label representing its memory address: s[0][0], s[0][1], s[0][2], s[0][3], s[0][4] for the first row; s[1][0], s[1][1], s[1][2], s[1][3], s[1][4] for the second row; and s[2][0], s[2][1], s[2][2], s[2][3], s[2][4] for the third row.

s[0][0]	s[0][1]	s[0][2]	s[0][3]	s[0][4]
s[1][0]	s[1][1]	s[1][2]	s[1][3]	s[1][4]
s[2][0]	s[2][1]	s[2][2]	s[2][3]	s[2][4]



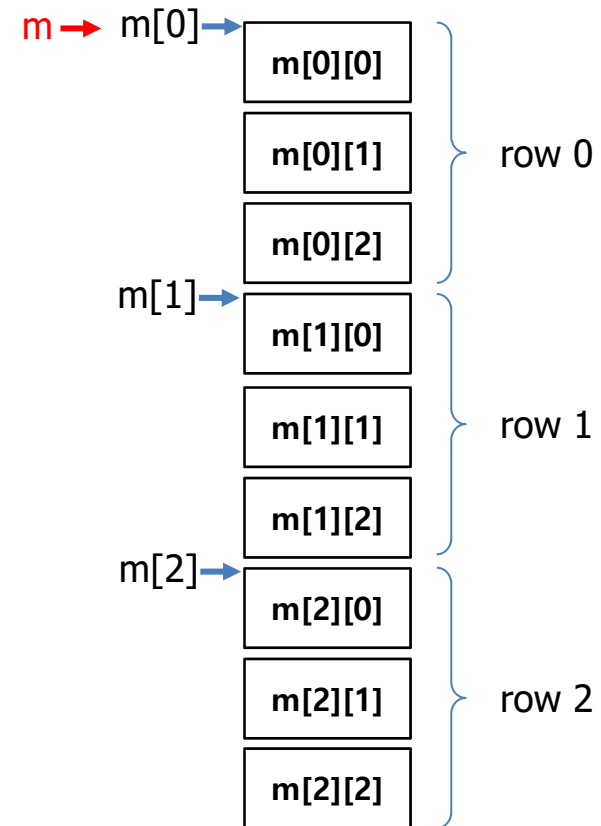
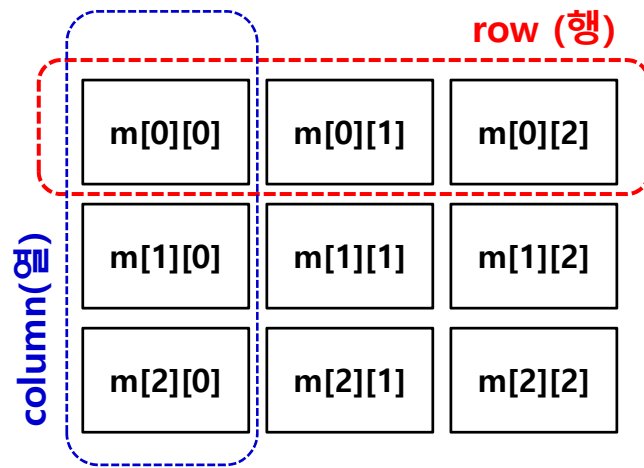
2차원 배열의 초기화

```
int s[3][5] = {  
    { 0, 1, 2, 3, 4 }, // 첫 번째 행의 원소들의 초기값  
    { 10, 11, 12, 13, 14 }, // 두 번째 행의 원소들의 초기값  
    { 20, 21, 22, 23, 24 } // 세 번째 행의 원소들의 초기값  
};
```



2차원 배열과 포인터

- ◆ 2차원 배열 `int m[3][3]`
- ◆ row 0, row 1, row 2 ... 순으로 메모리에 저장 (행 우선 방법)



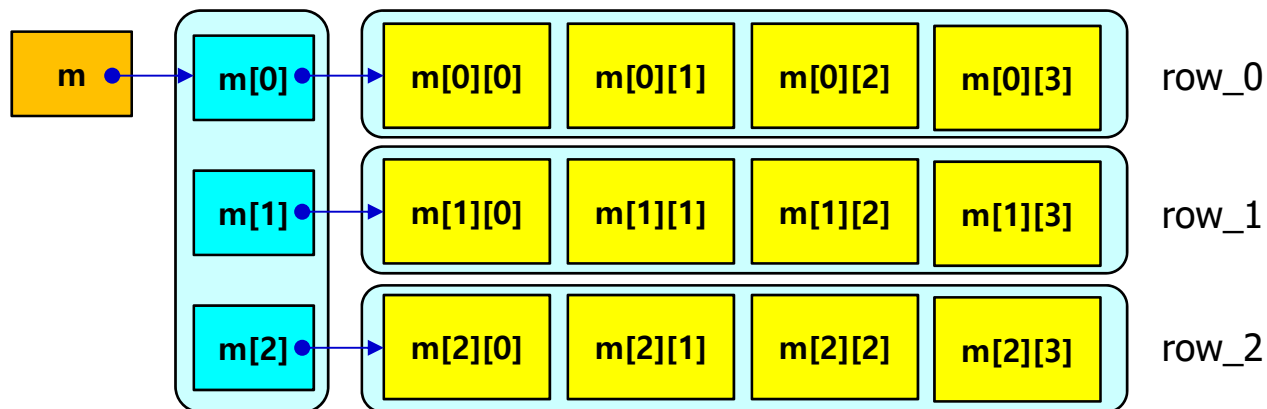
2차원 배열과 포인터

- ◆ 배열 이름 m 은 $\&m[0][0]$
- ◆ $m[0]$ 는 row 0의 시작 주소
- ◆ $m[1]$ 은 row 1의 시작 주소
- ◆ $m[2]$ 는 row 2의 시작 주소

double m[3][4];

m 은 3개의 1 차원 배열을 원소로 가지는 2차원 배열이다.

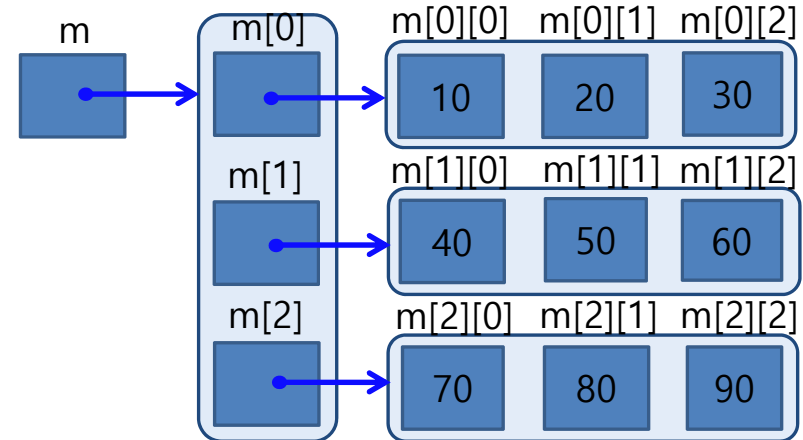
$m[]$ 의 각 원소는 4개의 double 형 데이터를 가지는 1차원 배열이다.



2 차원 배열 원소의 주소

```
void checkAddress_2DArray_for_Matrix(FILE *fout)
{
    int m[3][3] =
        { 10, 20, 30, 40, 50, 60, 70, 80, 90 };

    printf("m      = %p\n", m);
    printf("m[0]   = %p\n", m[0]);
    printf("m[1]   = %p\n", m[1]);
    printf("m[2]   = %p\n", m[2]);
    printf("&m[0][0] = %p\n", &m[0][0]);
    printf("&m[1][0] = %p\n", &m[1][0]);
    printf("&m[2][0] = %p\n", &m[2][0]);
    printf("\n");
}
```



<code>m</code>	<code>= 0036F988</code>
<code>m[0]</code>	<code>= 0036F988</code>
<code>m[1]</code>	<code>= 0036F9C4</code>
<code>m[2]</code>	<code>= 0036F9D0</code>
<code>&m[0][0]</code>	<code>= 0036F988</code>
<code>&m[1][0]</code>	<code>= 0036F9C4</code>
<code>&m[2][0]</code>	<code>= 0036F9D0</code>

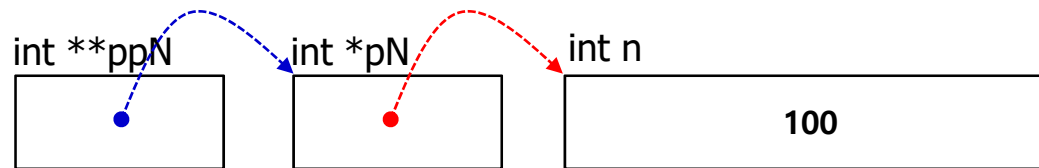
동일한 주소 !!



이중 포인터 (double pointer)

◆ 이중 포인터(double pointer) : 포인터를 가리키는 포인터

```
int n = 100;           // n는 int 형 변수  
int *pN = &n;          // pN는 n를 가리키는 포인터  
int **ppN = &pN;       // ppN는 포인터 pN를 가리키는 이중 포인터 (double pointer)
```

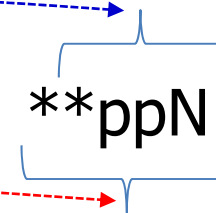


pN : int n의 주소

*pN : pN가 가리키는 위치의 데이터 (n)

*ppN : ppN이 가리키는 위치의 데이터 (pN의 값, n의 주소)

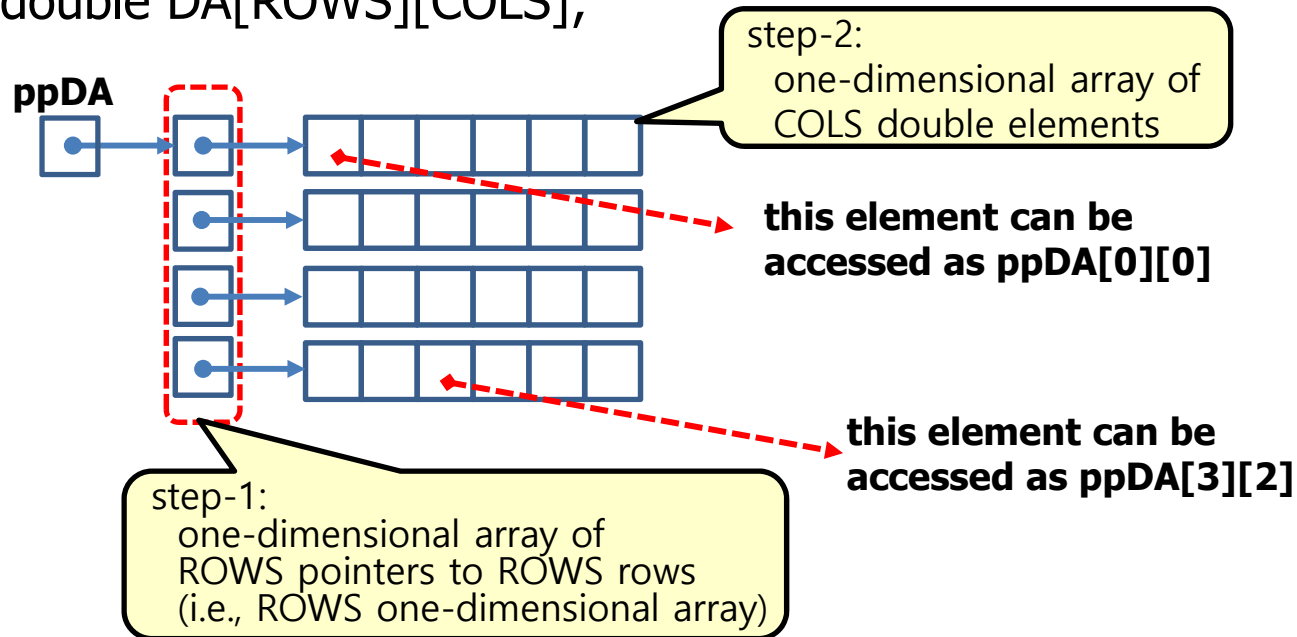
**ppN : ppN가 가리키는 위치의 데이터 (pN의 값, n의 주소)가
가리키는 위치의 데이터 (n)



2차원 배열의 동적 생성

◆ 2-dimensional Dynamic Array of double type (1)

- `double DA[ROWS][COLS];`



- can be considered as four rows of one dimensional array: `DA[i][0..4]`
- each row is pointed by a pointer to 1-dimension array:
- Array of pointers to the four 1-dimension arrays of pointers
 - **`double **ppDA = (double **)calloc(ROWS, sizeof(double *));`**
- To make the 2-dimensional array
 - **`ppDA[i] = (double *)calloc(COLS, sizeof(double));`**

2차원 배열의 동적 생성

◆ 2차원 실수 (double) 배열의 동적 생성 함수

```
double **create2DimDoubleArray(int row_size, int col_size)
{
    double **ppDA;

    ppDA = (double **)calloc(row_size, sizeof(double *));
    for (int r = 0; r < row_size; r++)
    {
        ppDA[r] = (double *)calloc(col_size, sizeof(double));
    }

    return ppDA;
}
```



2차원 배열 관련 기본 함수

◆ Mtrx.h

```
/* Mtrx.h */
#ifndef MTRX_H
#define MTRX_H

double **create2DimDoubleArray(int row_size, int col_size);
void delete2DimDoubleArray(double **dM, int row_size, int col_size);
void fget2DimDoubleArray(FILE *fp, double **dM, int row_size, int col_size);
void print2DimDoubleArray(double **dM, int row_size, int col_size);
double getRow_Avg(double **dM, int row, int num_cols);
double getTotal_Avg(double **dM, int num_rows, int num_cols);

#endif
```



```

/* Mtrx.cpp (1) */
#include <stdio.h>
#include <stdlib.h>
#include "Mtrx.h"

double **create2DimDoubleArray(int row_size, int col_size)
{
    double **ppDA;

    ppDA = (double **)calloc(row_size, sizeof(double *));
    for (int r = 0; r < row_size; r++)
    {
        ppDA[r] = (double *)calloc(col_size, sizeof(double));
    }

    return ppDA;
}

void delete2DimDoubleArray(double **dM, int row_size, int col_size)
{
    for (int r = 0; r < row_size; r++)
    {
        free(dM[r]);
    }
    free(dM);
}

```



```
/* Mtrx.cpp (2) */
```

```
void fget2DimDoubleArray(FILE *fp, double **dM)
```

```
{
    double data = 0.0;
    int row_size, col_size;

    if (fp == NULL)
    {
        printf("Error in get2DimDoubleArray() - file pointer is NULL !!\n");
        exit(-1);
    }
    fscanf(fp, "%d %d", &row_size, &col_size);
    for (int r = 0; r < row_size; r++)
        for (int c = 0; c < col_size; c++)
        {
            if (fscanf(fp, "%lf", &data) != EOF)
                dM[r][c] = data;
        }
}
```

```
void print2DimDoubleArray(double **mA, int row_size, int col_size)
```

```
{
    for (int r = 0; r < row_size; r++) {
        for (int c = 0; c < col_size; c++)
        {
            printf("%8.2lf", mA[r][c]);
        }
        printf("\n");
    }
}
```



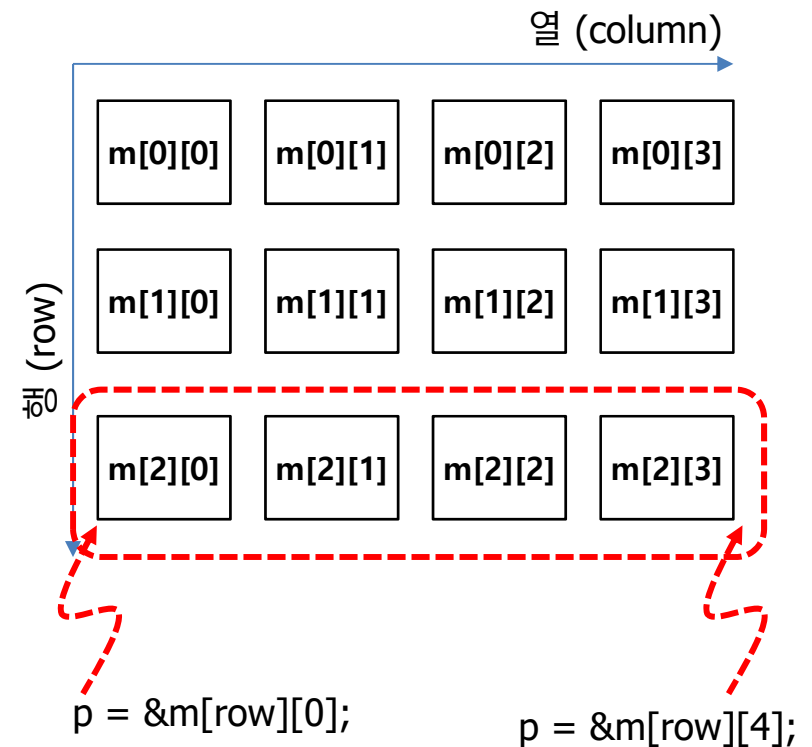
포인터를 이용한 배열 원소 방문

◆ 행(row)의 평균을 구하는 경우

```
/* Mtrx.cpp (3) */
```

```
double getRow_Avg(double **array,  
                  int row, int col_size)
```

```
{  
    double sum = 0.0, row_avg;  
  
    for (int c = 0; c < col_size; c++)  
    {  
        sum += array[row][c];  
    }  
  
    row_avg = sum / (double) col_size;  
    return row_avg;  
}
```



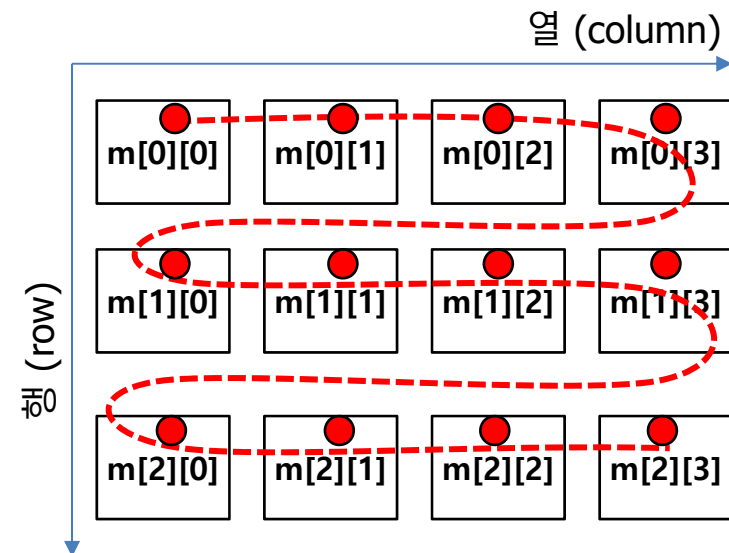
포인터를 이용한 배열 원소 방문

◆ 2차원 배열 전체 원소의 평균을 구하는 경우

```
/* Mtrx.cpp (4) */
```

```
double getTotal_Avg(double **dM,  
    int row_size, int col_size)
```

```
{  
    double sum = 0.0, avg;  
  
    for (int r = 0; r < row_size; r++)  
    {  
        for (int c = 0; c < col_size; c++)  
        {  
            sum += dM[r][c];  
        }  
    }  
  
    avg = sum / (double)(row_size * col_size);  
    return avg;  
}
```



Homework 7

Homework 7

7.1 함수의 인수 전달 방식에 대한 비교

- 두 개의 double 자료형 데이터를 함수의 인수로 전달하고, 그 평균값을 계산하여 double 자료형으로 반환하는 간단한 average() 함수를 call-by-value, call-by-pointer, call-by-reference 방식으로 각각 구현하고, 정상적으로 실행되는 것을 확인하라.

`double average_value(double x, double y); // call-by-value`

`void average_pointer(double *px, double *py, double *pavg); // call-by-pointer`

`void average_reference(double &x, double &y, double &avg); // call-by-reference`

- 이 세가지 함수 인수 전달 방법의 장점과 단점을 비교하여 설명하라.



Homework 7

7.2 파일 입력, 2차원 배열의 동적 생성, 행렬의 덧셈, 뺄셈, 곱셈 연산

- 행렬의 행의 크기와 열의 크기를 나타내는 정수 데이터 2개와 (행의 크기 x 열의 크기) 개수의 double 자료형 데이터를 포함하고 있는 입력 데이터 파일 (input_data.txt)를 읽어 2차원 배열을 동적으로 생성하는 함수 void inputMtrx(FILE *fin, double **ppM, int *row_size, int *col_size)를 작성하라. row_size와 col_size는 return-by-pointer 방식으로 파일로 부터 입력된 값을 전달할 것.
- 입력파일의 예:
- 동적으로 생성된 2차원 배열 (행렬)을 소멸시키는 함수 void deleteMtrx(double **ppM, int row_size, int col_size)를 작성하라.
- 행렬의 입력 및 동적 생성, 소멸을 위한 함수들은 Mtrx.cpp에 구현하고, 해당 함수 원형은 Mtrx.h 헤더파일에 포함시킬 것.

```
5 5
1.1 2.2 3.3 4.4 5.5
6.6 7.7 8.8 9.9 10.0
11.1 12.2 13.3 14.4 15.5
16.6 17.7 18.8 19.9 20.0
21.1 22.2 23.3 24.4 25.5
```

```
5 5
1.1 0.0 0.0 0.0 0.0
0.0 2.2 0.0 0.0 0.0
0.0 0.0 3.3 0.0 0.0
0.0 0.0 0.0 4.4 0.0
0.0 0.0 0.0 0.0 5.5
```



Homework 7

7.2 파일 입력, 2차원 배열의 동적 생성, 행렬의 덧셈, 뺄셈, 곱셈 연산 (계속)

- 행렬을 전달받아 출력하는 함수 `void printMtrx(double **ppM, int row_size, int col_size)`를 작성하라. 이 때, 행렬을 대괄호로 표시하기 위하여, 확장 완성형 코드를 사용할 것.
각 배열 원소의 값은 소숫점 이하 2자리까지 출력할 것.

- 출력 예시:

```
Matrix A :  
[ 1.10  2.20  3.30  4.40  5.50  
  6.60  7.70  8.80  9.90 10.00  
 11.10 12.20 13.30 14.40 15.50  
 16.60 17.70 18.80 19.90 20.00  
 21.10 22.20 23.30 24.40 25.50]
```

- 2개의 행렬 A, B를 전달받아 덧셈, 뺄셈, 곱셈을 계산하여 그 결과를 2차원 동적 배열을 생성하여 반환하는 행렬 연산 함수를 작성하라.
 - `double ** addMtrx(double **A, double **B, int row_size, int col_size);`
 - `double ** subMtrx(double **A, double **B, int row_size, int col_size);`
 - `double ** mulMtrx(double **A, double **B, int row_size, int col_size);`
- 행렬의 덧셈, 뺄셈, 곱셈 연산 함수는 `Mtrx.cpp` 파일에 포함시키고, 함수 원형은 `Mtrx.h` 헤더파일에 포함시킬 것.



Homework 7

7.2 파일 입력, 2차원 배열의 동적 생성, 행렬의 덧셈, 뺄셈, 곱셈 연산 (계속)

- 파일로 부터 행렬 A와 B를 입력받고, 이 행렬의 덧셈, 뺄셈, 곱셈을 실행한 후, 그 결과를 출력하는 main() 함수를 작성하라.
- 행렬 연산을 총괄하는 main() 함수에서는 파일로 부터 행렬을 입력받아 동적으로 생성하며, printMtrx(), addMtrx(), subMtrx(), mulMtrx() 함수를 호출하여, 그 결과를 출력하여야 함.
- main()의 마지막 부분에서는 동적으로 생성된 행렬들을 소멸시키는 기능이 있어야 하며, 입력 파일을 닫는 기능이 포함되어야 함
- 실행 결과 (예시)

```
Matrix A :  
[ 1.10  2.20  3.30  4.40  5.50  
  6.60  7.70  8.80  9.90 10.00  
 11.10 12.20 13.30 14.40 15.50  
 16.60 17.70 18.80 19.90 20.00  
 21.10 22.20 23.30 24.40 25.50  
Matrix B :  
[ 1.10  0.00  0.00  0.00  0.00  
  0.00  2.20  0.00  0.00  0.00  
  0.00  0.00  3.30  0.00  0.00  
  0.00  0.00  0.00  4.40  0.00  
  0.00  0.00  0.00  0.00  5.50  
Matrix C = A + B:  
[ 2.20  2.20  3.30  4.40  5.50  
  6.60  9.90  8.80  9.90 10.00  
 11.10 12.20 16.60 14.40 15.50  
 16.60 17.70 18.80 24.30 20.00  
 21.10 22.20 23.30 24.40 31.00  
Matrix D = A - B:  
[ 0.00  2.20  3.30  4.40  5.50  
  6.60  5.50  8.80  9.90 10.00  
 11.10 12.20 10.00 14.40 15.50  
 16.60 17.70 18.80 15.50 20.00  
 21.10 22.20 23.30 24.40 20.00  
Matrix E = A * B:  
[ 1.21  4.84 10.89 19.36 30.25  
  7.26 16.94 29.04 43.56 55.00  
 12.21 26.84 43.89 63.36 85.25  
 18.26 38.94 62.04 87.56 110.00  
 23.21 48.84 76.89 107.36 140.25
```

