# 10. Arrays and Pointers

### Keyword: static

### Operators: & \* (unary)

### arrays을 생성하고 초기화하는 방법

### Pointers와 arrays를 연관시켜 사용하는 방법

### arrays을 처리하는 function을 작성하기

### 2차원 arrays

## 10.1 Arrays

### array은 한가지 data type의 elements의 series로 구성됨

/\* 어떤 array 선언 \*/

int main(void)

{

float candy[365]; /\* array of 365 floats \*/

char code[12]; /\* array of 12 chars \*/

int states[50]; /\* array of 50 ints \*/

...

}

### array을 접근하기 위하여, index라 불리는 첨자 숫자(subscript number)를 사용하여 개별 element를 identify한다.

## 10.1.1 Initialization

### single-valued variables 초기화 (scalar variables라고 부름)

int fix = 1;

float flax = PI \* 2;

### arrays의 초기화

int main(void)

{

int powers[8] = {1,2,4,6,8,16,32,64}; /\* ANSI C and later \*/

...

}

<<listing 10.1>>

### array를 초기화하지 않을 경우에는? Listing 10.2

### 배열 일부만 초기화하는 경우? Listing 10.3

#### 나머지 elements는 0으로 초기화

### array를 초기화할 때 empty bracket 사용시

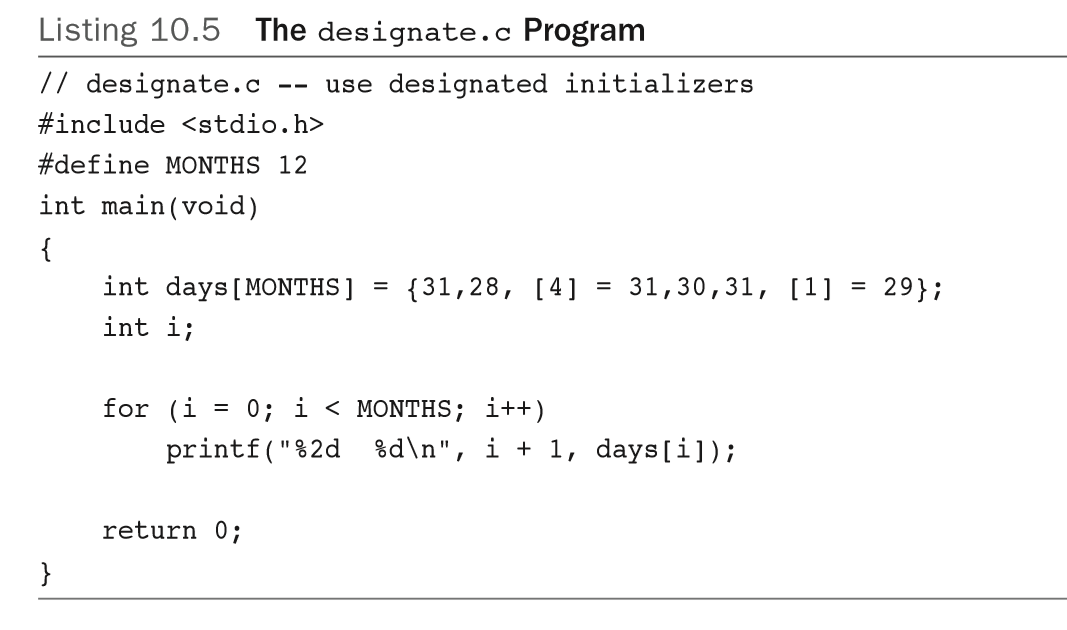
#### 초기화 list에 있는 items 숫자를 세고 array의 크기를정함 (Listing 10.4 ).

## 10.1.2 Designated Initializers (C99) – Visual Studio 2015는 c99를 지원하지 않음

### 특정 element를 초기화 지정하기 위해 initialization list에서 bracket index를 사용:

int arr[6] = {0,0,0,0,0,212}; // 전통적 방법

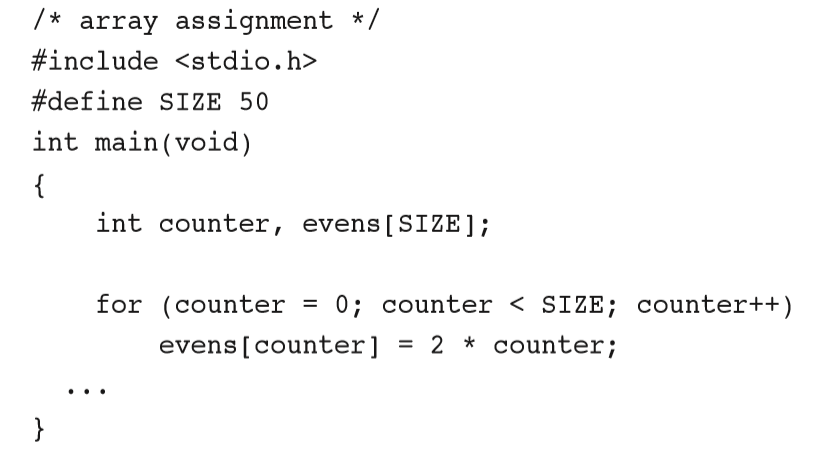
int arr[6] = {[5] = 212}; //arr[5]을 212로 초기화



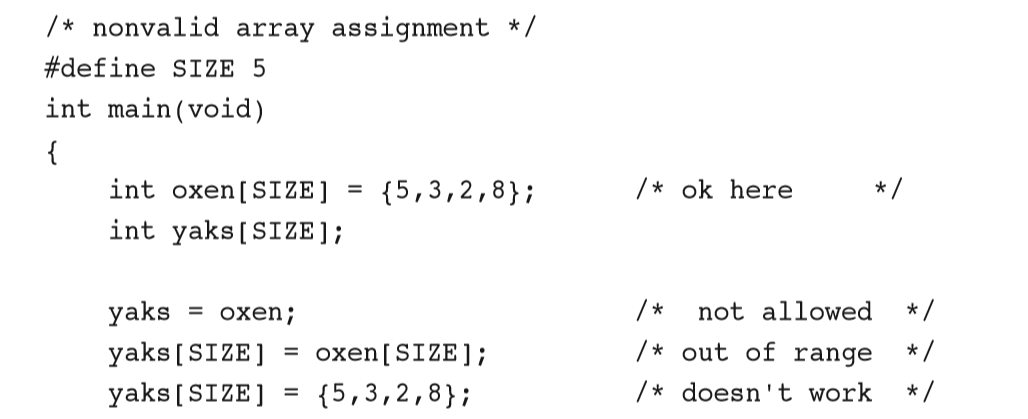
##### Visual Studio2019는 지원하지 않음!

10.1.3 Array 값으로 assign하기

### array index, subscript를 사용하여 array member에 값을 assign



### C는 하나의 array를 한 unit으로 다른 array로 assign하지 못함



## 10.1.4 Array Bounds

### compiler는 indices가 valid한지 check하지 않음

### Listing 10.6: array의 bound를 check

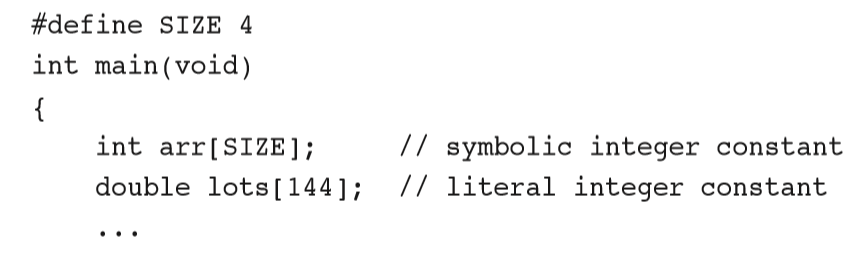
#### index 값을 -1 ~ 6 값 사용.

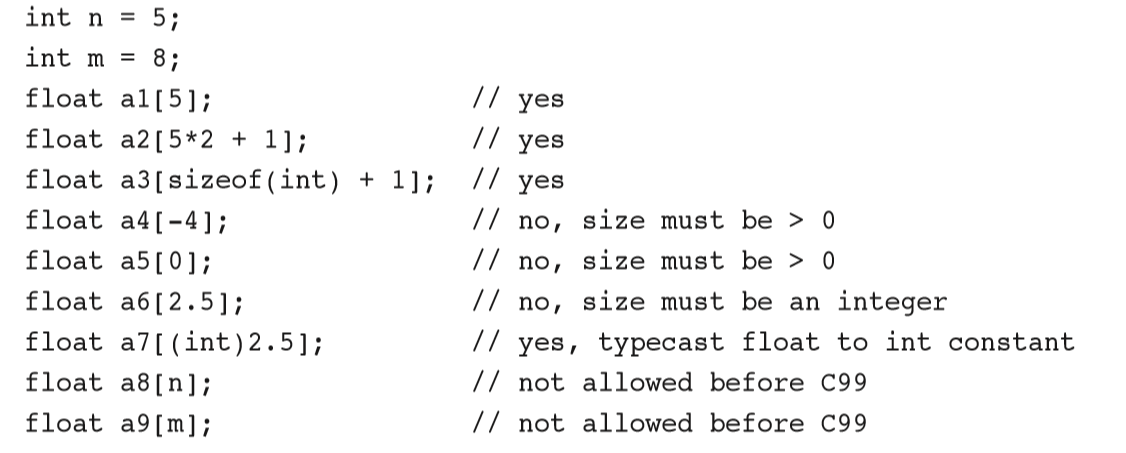
#### C 철학: C programmer에게 trust.

#### Not checking bounds가 C program이 더 빨리 동작하게 한다

### C는 프로그래머가 코딩을 정확히 할 것을 trust한다. 보답으로 faster program을 제공한다

## 10.1.5 Array Size 지정





### C99는 가변 길이 array를 지원

## 10.2 다차원 Arrays

### better approach는 array of array를 사용하는 것

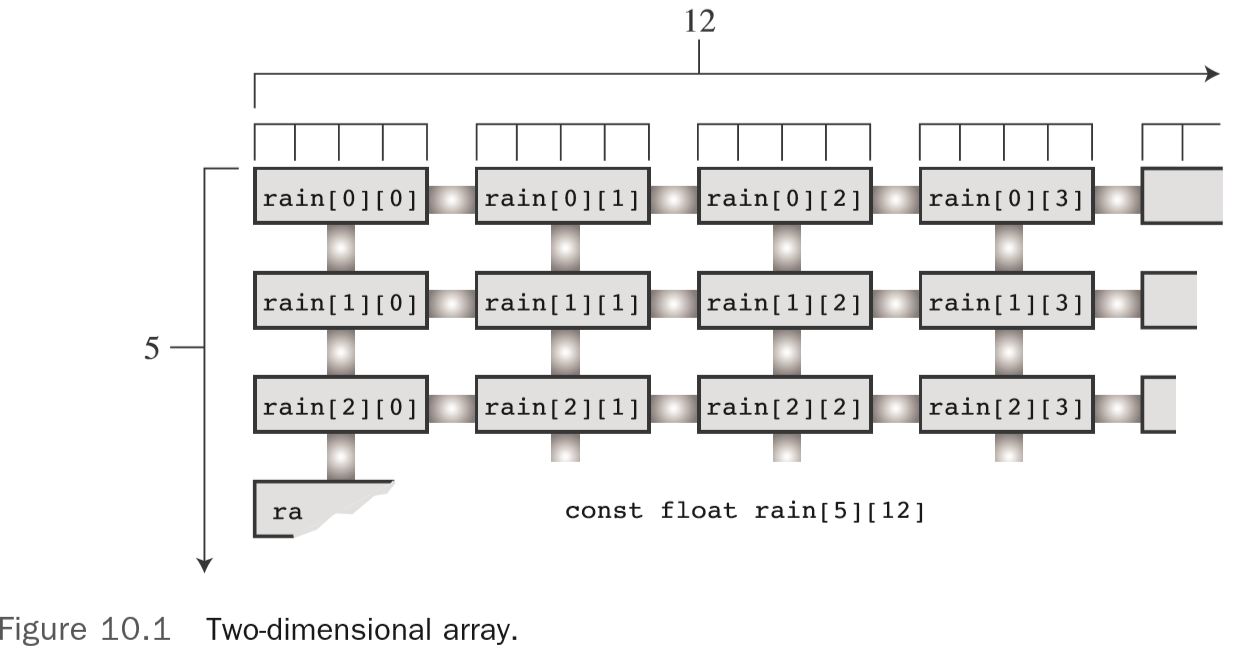
float rain[5][12]; //5 arrays of 12 floats array

### rain[0], rain의 첫번째 element: an array of 12 float values.

### rain은 5-element array of 12-element arrays of float ,

#### rain[0]은 array of 12 floats

#### rain[0][0]은 float

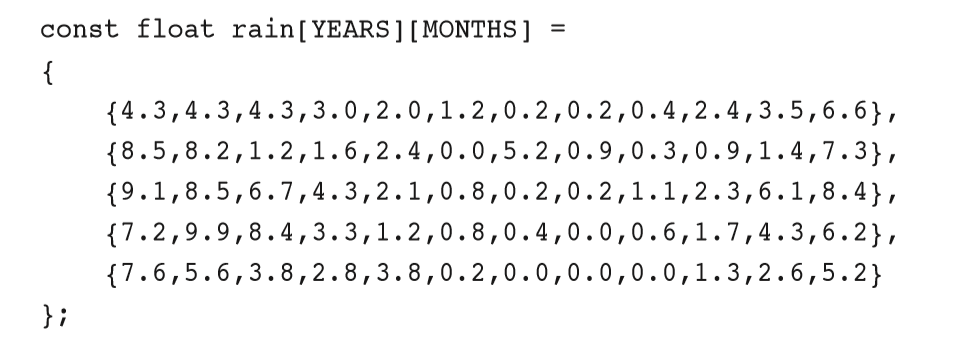


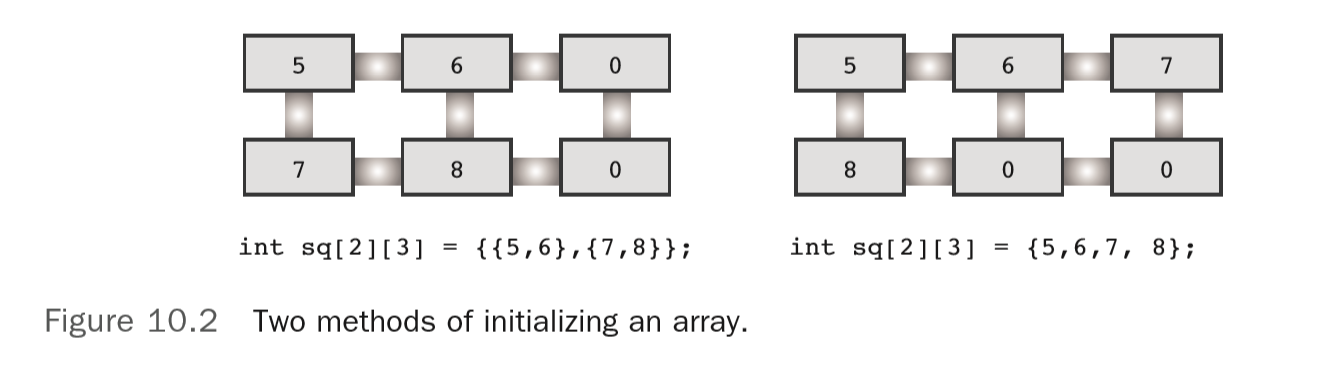
### listing 10.7: rain.c program

## 10.2.1 2차원 Array의 초기화

### 1차원 array의 초기화

sometype ar1[5] = {val1, val2, val3, val4, val5};





## 10.2.2 3차원 array

int box[10][20][30];

### box array는 10개의 2차원 arrays (each 20×30)

## 10.3 Pointers and Arrays

flizny == &flizny[0]; // array 이름 = 첫째 element의 address

* printf("pointers: %10p %10p %10p\n", dates, &dates, &dates[0]);

### flizny 와 &flizny[0]는 첫째 element의 memory address

#### & : address operator.

##### array name은 constants (program 실행 동안에 변경되지 않는다)

#### 주소 값은 pointer variable에 assign될 수 있다, pointer variable은 나중에 변경될 수 있다

<<Listing 10.8>>

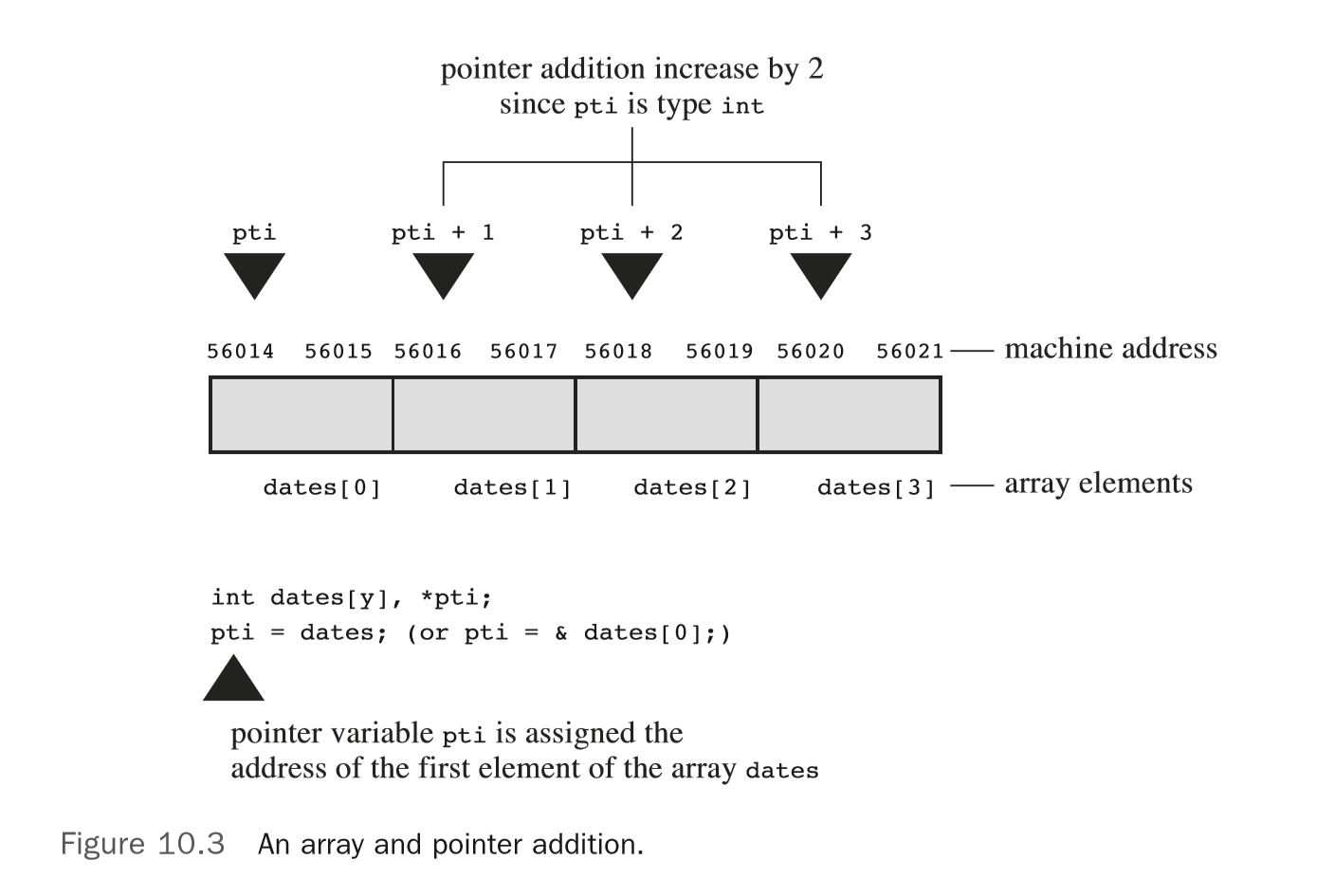
### pointer variable의 값의 변경: “add 1 to a pointer,”

#### C adds one storage unit .

#### arrays에 대하여 address가 next element의 address로 변경, not just the next byte

#### Int days[12] = {…};

#### printf("\nMonth : %10p %10p %10p\n", days, days+1, days+2);



### pointer 값은 point하는 object의 address

### \* operator: pointed-to object에 저장된 값을 준다

### pointer에 1 증가: pointed-to type의 size(byte로) 만큼 주소 값을 증가시킨다

dates + 2 == &date[2] // 같은 address

\*(dates + 2) == dates[2] // 같은 값

#### ar[n] == \*(ar + n) .

#### \*(dates+2) != \*dates+2 .

##### \*(dates + 2) // dates의 3rd element 값

##### \*dates + 2 // 1st element의 값에 2를 증가시킴

<<Listing 10.9>>

printf("\nMonth : %10p %10p %10p\n", days, days+2, &days[2]);

printf(“Values: %d %d %d”, \*(days+2), days[2], \*days +2)

## 10.4 Functions, Arrays, and Pointers

### array에 대하여 동작하는 function의 작성

### array elements의 합을 return하는 function 작성

total = sum(marbles); // possible function call

### array를 전달하는 function protype?

#### array name은 first element의 address,

##### actual argument marbles은 int 객체의 address로서, pointer-to- int인 formal parameter에 assign된다

int sum(int \* ar); // 상응하는 prototype

##### sum()은 argument로부터 가져오는 정보는?

##### array의 1st element의 address를 얻게 된다. 해당 location에 있는 int 값을 find한다

##### //Listing 10.10 The sum\_arr1.c Program

##### // sum\_arr1.c -- 함수에 배열 전달 - call by pointer

int sum(int ar[], int n) // array는 얼마나 큰가?

{

int i;

int total = 0;

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

total += ar[i];

return total;

}

int sum(int \*ar, int n) //array는 얼마나 큰가?

{

int i;

int total = 0;

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

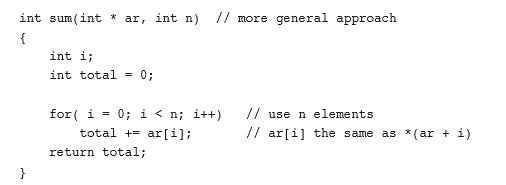
total += ar[i];//total += \*(ar+i);

return total;

}

### array names으로 pointer notation을 사용할 수 있다

### 좀 더 유연한 approach는 array size를 두번째 argument로 전달하는 것이다



### int ar[] 대신에 int \* ar을 사용할 수 있다 :

int sum (int ar[], int n);

### int \* ar 표기는 ar 이 pointer-to-int라는 것을 의미한다.

### int ar[] 표기는 ar 이 pointer-to- int라는 것을 의미한다

##### 다음 4가지 표현은 모두 동일하다:

int sum(int \*ar, int n);

int sum(int \*, int);

int sum(int ar[], int n);

int sum(int [], int);

##### 다음 두가지 표현은 동일하다:

int sum(int \*ar, int n) {

// code goes here }

int sum(int ar[], int n); {

// code goes here }

## 10.4.1 Using Pointer Parameters

### sum() function은 array의 첫번째 element의 주소를 식별하기 위하여 pointer parameter를 사용한다. 정수 parameter 는 얼마나 많은 elements를 처리할 것인지를 알려주는 것이다.

//Listing 10.11 The sum\_arr2.c Program

/\* sum\_arr2.c -- sums the elements of an array \*/

#include <stdio.h>

#include <stdlib.h>

#define SIZE 10

int sump(int\* start, int\* end);

int main(void)

{

int marbles[SIZE] = { 20,10,5,39,4,16,19,26,31,20 };

long answer;

answer = sump(marbles, marbles + SIZE);

printf("marble의 전체갯수는 %ld.\n", answer);

system("pause");

return 0;

}

/\* use pointer arithmetic \*/

int sump(int\* start, int\* end)// sump(int start[], int end[])와 동일하다

{

int total = 0;

while (start < end)

{

total += \*start; // add value to total

printf("\*start = %d\n", \*start);

start++; // advance pointer to next element

}

return total;

}

#### start++ 표현은 공통적으로 많이 사용된다,

//Listing 10.12 The order.c Program

/\* order.c -- precedence in pointer operations \*/

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

int data[2] = { 100, 200 };

int moredata[2] = { 300, 400 };

int\* p1, \* p2, \* p3;

p1 = p2 = data;

p3 = moredata;

printf(" \*p1 = %d, \*p2 = %d, \*p3 = %d\n", \*p1, \*p2, \*p3);

printf("\*p1++ = %d, \*++p2 = %d, (\*p3)++ = %d\n", \*p1++, \*++p2, (\*p3)++);

printf(" \*p1 = %d, \*p2 = %d, \*p3 = %d\n", \*p1, \*p2, \*p3);

/\*

printf("data: %d - %d", \*data, \*(data + 1), \*(data++));//식이 수정할 수 있는 l-value가 아니다

printf("data: %d - %d", \*data, \*(data + 1), \*data++);

\*/

system("pause");

return 0;

}

## 10.5 Pointer Operations

### pointer에 대하여 실행할 수 있는 operations

//Listing 10.13 The ptr\_ops.c Program

// ptr\_ops.c -- pointer operations

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

int urn[5] = { 100,200,300,400,500 };

int\* ptr1, \* ptr2, \* ptr3;

ptr1 = urn; // assign an address to a pointer

ptr2 = &urn[2]; // dereference a pointer and take

// the address of a pointer

printf("pointer value, dereferenced pointer, pointer address:\n");

printf("ptr1 = %p, \*ptr1 =%d, &ptr1 = %p\n", ptr1, \*ptr1, &ptr1);//ptr1, &ptr1의 주소 값이 다르다

// pointer addition

ptr3 = ptr1 + 4;

printf("\nadding an int to a pointer:\n");

printf("ptr1 + 4 = %p, \*(ptr4 + 3) = %d\n", ptr1 + 4, \*(ptr1 + 3));//ptr1 + 4가 가리키는 주소를 확인

ptr1++; // increment a pointer

printf("\nvalues after ptr1++:\n");

printf("ptr1 = %p, \*ptr1 =%d, &ptr1 = %p\n", ptr1, \*ptr1, &ptr1);//ptr1, &ptr1의 주소 값의 변화를 확인

ptr2--; // decrement a pointer

printf("\nvalues after --ptr2:\n");

printf("ptr2 = %p, \*ptr2 = %d, &ptr2 = %p\n", ptr2, \*ptr2, &ptr2);//ptr2, &ptr2의 주소 값의 변화를 확인

--ptr1; // restore to original value

++ptr2; // restore to original value

printf("\nPointers reset to original values:\n");

printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);

// subtract one pointer from another

printf("\nsubtracting one pointer from another:\n");

printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %td\n", ptr2, ptr1, ptr2 - ptr1);

// subtract an integer from a pointer

printf("\nsubtracting an int from a pointer:\n");

printf("ptr3 = %p, ptr3 - 2 = %p\n", ptr3, ptr3 - 2);

system("pause");

return 0;

}

#### Assignment— pointer에 address를 assign할 수 있다

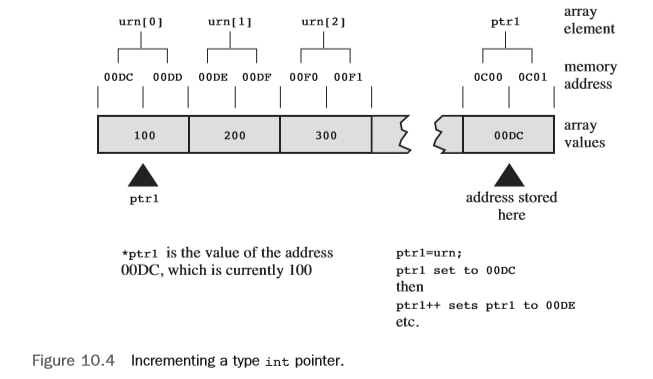
#### Value finding (dereferencing)— \* operator는 pointed-to location에 저장된 값을 준다

#### pointer 변수가 address를 값으로 갖는다— 모든 variable처럼, pointer variable도 값으로 address를 갖는다

#### pointer에 정수를 더하기— + operator를 사용하여 pointer에 정수를 더 할 수 있다.

#### pointer의 increment —

##### pointer to array의 증가는 array의 next element로 이동하는 것이다



#### pointer로부터 정수를 빼기 —

##### operator를 사용하여 pointer로부터 정수를 빼기를 할 수 있다

#### pointer의 decrement—

##### pointer의 decrement는 array의 앞 element로 이동하는 것이다

#### Differencing— 두개의 pointer 값의 차이를 계산할 수 있다

#### Comparisons— relational operators를 사용하여 두개의 pointer의 값을 비교할 수 있다, 단 두개의 pointer 타입은 같은 type 일 것.

## 10.5.1 초기화되지 않은 pointer의 dereferencing

int \* pt; // 초기화되지 않은 pointer

\*pt = 5; // a terrible error

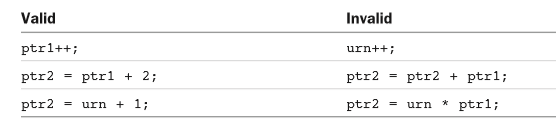
### Run-Time Check Failure #3 - The variable 'pt' is being used without being initialized.

### 이 문제가 심각한 이유는? “\*pt = 5;”은 pt가 point하는 location에 5를 저장하라는 것, 그런데 pt가 가리키는 곳이 없음.

#### pt는 초기화되지 않아 임의 숫자를 가지게 됨, 따라서 5를 저장할 메모리 공간 상의 위치를 모르게 됨

int urn[3];

int \* ptr1, \* ptr2;



### C programmers는 arrays of pointers, pointers to functions, arrays of pointers to pointers, arrays of pointers to functions을 만든다

### Pointer의 첫번째 기본적 사용은 function의 parameter 전달이다

#### 호출된 function에서의 변수 값 변경이 main에서의 변수 값의 변경으로 영향을 주려면 parameter 전달시에 pointers를 사용해야 한다.

### Pointer 의 두번째 사용은 array를 처리하는 functions에서 사용한다

//Listing 10.14 The arf.c Program

/\* arf.c -- array functions \*/

#include <stdio.h>

#include <stdlib.h>

#define SIZE 5

void show\_array(const double ar[], int n);

void mult\_array(double ar[], int n, double mult);

int main(void)

{

double dip[SIZE] = { 20.0, 17.66, 8.2, 15.3, 22.22 };

printf("The original dip array:\n");

show\_array(dip, SIZE);

mult\_array(dip, SIZE, 2.5);

printf("The dip array after calling mult\_array():\n");

show\_array(dip, SIZE);

system("pause");

return 0;

}

/\* displays array contents \*/

void show\_array(const double ar[], int n)

{

int i;

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

// printf("%8.3f ", ar[i]);

printf("%8.3f ", \*(ar+i));//pointer로 사용한다

putchar('\n');

}

/\* multiplies each array member by the same multiplier \*/

void mult\_array(double ar[], int n, double mult)

{

int i;

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

\*(ar+i) \*= mult;

}

## 10.6 Protecting Array Content

### call by value:

#### function에 parameter 전달 기본 원칙: pass by value

#### C function은 pass by value가 기본: to preserve the integrity of the data

##### Function이 array을 전달할 때 array의 address가 value 로서 전달되며, function의 formal parameter는 pointer로서 받아야 한다.

### array-processing functions은 main()의 array의 element 를 변경할 수 있다

void add\_to(double ar[], int n, double val)

{

int i;

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

ar[i] += val;

}

### 다음 function은 array’s contents의 sum을 구하는 것이 목적이고 array의 element를 변경하지 않아야 하는데 논리적 오류시 발견 못함.

#### Sum() 작성자가 잘 못 프로그래밍시에 main()의 array를 변경하게 되는 문제가 있다

int sum(int ar[], int n) // faulty code

{

int i;

int total = 0;

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

total += ar[i]++; // error increments each element

return total;

}

## 10.6.1 Formal Parameters에 const 사용

### function의 의도가 array의 내용을 변경하는 것이 아니면 formal parameter에 const 를 사용

### const를 사용한다고 해서 original array가 constant여야 한다는 것을 의미하는 것은 아니다

#### function는 array를 constant처럼 취급한다.

## 10.6.2 const에 관한 고급 지식

const double PI = 3.14159;

### #define directive,

#define MONTHS 12

### const를 사용하여 constant arrays, constant pointers, pointers to constants를 정의할 수 있다

const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};

days[9] = 44; /\* compile error \*/

### Pointers to constants는 값을 변경하는 데 사용될 수 없다

### double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};

### const double \* pd = rates; // pd는 array에 대한 pointer로서 pd 변수로 값을 변경하지 못한다는 의미이다 – 퀴즈 대상

\*pd = 29.89; // not allowed

pd[2] = 222.22; // not allowed

rates[0] = 99.99; // array rates는 const가 아니므로 허용됨

pd++; /\* make pd point to rates[1] -- allowed \*

#### pointer-to-constant로 formal parameter로 선언하면 해당 function이 pointer 를 사용하여 parameter를 변경하지 않는다는 것을 말한다

void show\_array(const double \*ar, int n);

#### pointer-to-constant 변수가 constant data or non-constant data의 address를 갖는 것은 문제없음

double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};

const double locked[4] = {0.08, 0.075, 0.0725, 0.07};

const double \* pc = rates; // valid

pc = locked; // valid

pc = &rates[3]; // valid

#### regular pointers 는 오직 non-constant data의 address를 값으로 가질 수 있다

double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};

const double locked[4] = {0.08, 0.075, 0.0725, 0.07};

double \* pnc = rates; // valid

pnc = locked; // not valid

pnc = &rates[3]; // valid

show\_array(rates, 5); // valid

show\_array(locked, 4); // valid

mult\_array(rates, 5, 1.2); // valid

mult\_array(locked, 4, 1.2); // bad idea

#### constant pointer: pointer 자체가 상수로서 변경이 불가하다

double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};

double \* const pc = rates; // pc는 array address를 갖는 상수 포인터

pc = &rates[2]; // pc 변경은 허용되지 않음

\*pc = 92.99; // ok -- changes rates[2] rates[0]

#### const pointer의 constant 값

double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};

const double \* const pc = rates;

pc = &rates[2]; // not allowed

\*pc = 92.99; // not allowed

## 10.7 Pointers and Multidimensional Arrays

### pointer를 사용하여 다차원 array를 처리하는 방법?

### 다차원 array를 처리하는 Function은 pointer를 사용하여 처리하는 프로그램으로 작성하는 것이 가능

int zippo[4][2]; /\* an array of arrays of ints \*

#### zippo는 &zippo[0]와 같은 값을 갖는다

#### zippo[0]는 an array of two integers,

#### zippo[0]는 &zippo[0][0]와 같은 값을 갖는다: 1번째 element(int)의 address

##### zippo[0]는 address of an int-sized object(**zippo[0]+1의 의미를 아는 것이 필요**)

##### zippo는 address of a two-int-sized object(**zippo+1의 의미를 아는 것이 중요**)

###### pointer 또는 address에 1 증가: referred-to object의 size 만큼 주소를 증가시킴

### **zippo + 1: 8 바이트 증가**//code 실행으로 확인하는 것이 필요

### **zippo[0] + 1: 4 바이트 증가**//code 실행으로 확인하는 것이 필요

#### Dereferencing a pointer or an address (pointer 변수에 \* operator 적용 또는 [] operator 적용하는 것)

//de- : do the opposite of, reverse of의 뜻, 역참조 번역은 무슨 뜻인지가 설명이 안됨, dereference는 참조되는 주소에 저장된 값을 가져오는 것: 반대 방향의 행위가 일어나는 것을 말함

### referred-to object 값을 준다.

### zippo[0][0] -> dereferencing이다

### \*(zippo[0]) -> dereferencing이다

### \*zippo, 또는 zippo[0]: address of an int

### \*\*zippo = \*&zippo[0][0]

#### address of an address 또는 pointer of a pointer : double indirection .

### zippo[0]: the address of its first element, ( zippo[0][0] ),

### \*(zippo[0]) : zippo[0][0]에 저장된 값

### \*zippo: zippo[0] 값과 같고 주소를 dereferencing

### \*zippo = zippo[0] = &zippo[0][0]

### \*\*zippo는 \*&zippo[0][0]와 같다

### zippo는 address of an address이다. 그리고 dereference를 두번해야 한다

//Listing 10.15 The zippo1.c Program

/\* zippo1.c -- zippo info \*/

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

const int n = 10;

//n++;

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

const int\* b = a; //pointers to constants

int\* p;

//b[1] = 4;

//\*b = 3;

\*a = 3;

a[1] = 2;

printf("\*b = %d\n", \*b);

b++;

printf("\*b = %d\n", \*b);

//p = b;//const int \* 의 값을 int \*에 assign할 수 없다

int c[4] = { 5,6,7,8 };

b = c;

double d[3] = { 1.1, 2.2, 3.3 };

double\* const e = d; //constant pointers

double\* ptr;

//e = &d[2];

\*e = 4.4;

printf("\*e = %f\n", \*e);

const double\* const f = d; //constant pointers to constants

//f = &d[1];//식이 수정할 수 있는 l-value가 아니다

//\*f = 5.5; //식이 수정할 수 있는 l-value가 아니다

ptr = d;

ptr = e;

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

printf(" zippo = %p, zippo + 1 = %p\n",

zippo, zippo + 1);

printf("zippo[0] = %p, zippo[0] + 1 = %p\n",

zippo[0], zippo[0] + 1);

printf(" \*zippo = %p, \*zippo + 1 = %p\n",

\*zippo, \*zippo + 1);

printf("zippo[0][0] = %d\n", zippo[0][0]);

printf(" \*zippo[0] = %d\n", \*zippo[2]);

printf(" \*\*zippo = %d\n", \*(zippo[1] + 1));

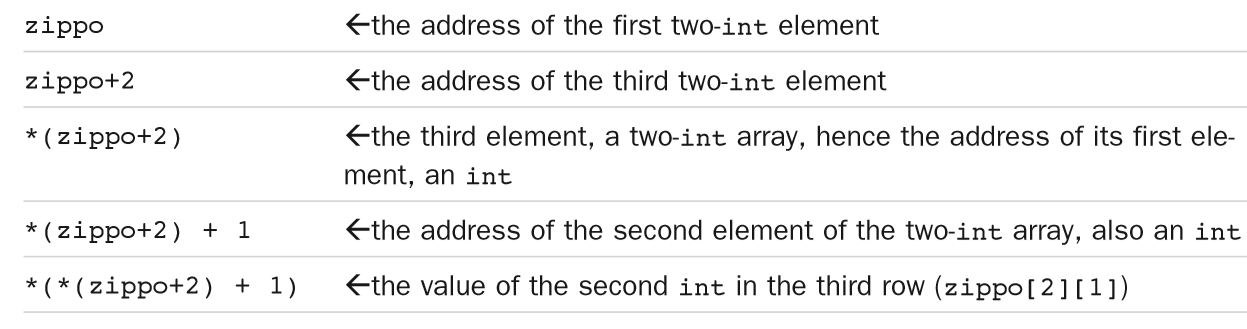
printf(" zippo[2][1] = %d\n", zippo[2][1]);

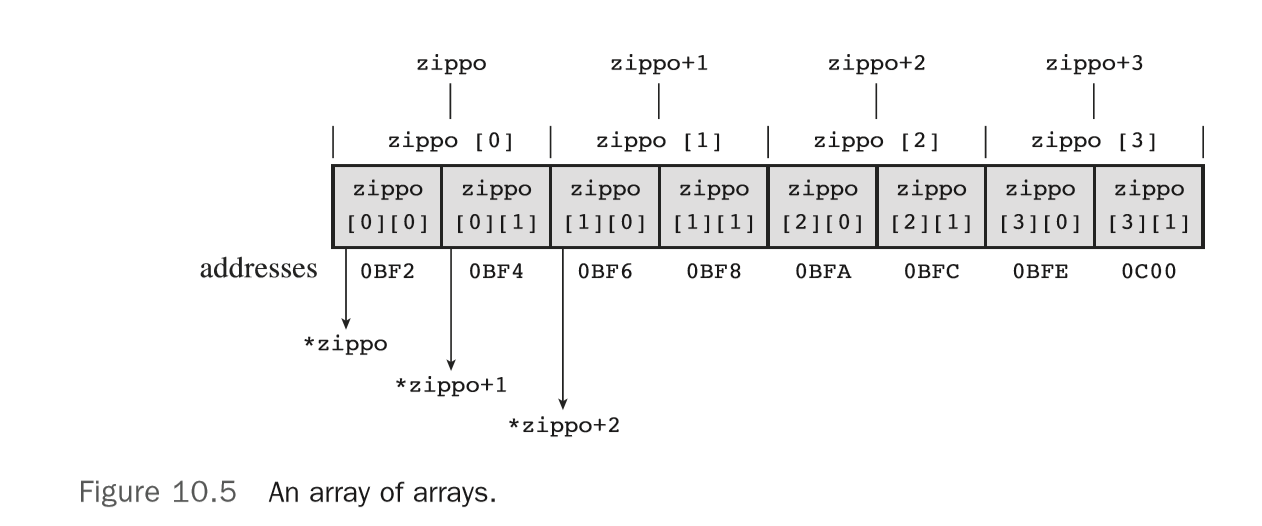
printf("\*(\*(zippo+2) + 1) = %d\n", \*(\*(zippo + 1) + 1));

system("pause");

return 0;

}





### adding 1 to zippo[0] : 주소가 4 증가

### adding 1 to zippo : 주소가 8 bytes 증가

### zippo[2][1]는 \*(\*(zippo+2) + 1)와 같다

## 10.7.1 Pointers to Multidimensional Arrays

## 10.7.1.1 Pointer to array: 배열 포인터?

int (\* pz)[2]; // pz는 array of 2 ints 를 point한다

### int zippo[3][2]; //zippo[0]도 pointer-to-int이며 pz 와 compatible 하다

### pz는 array of two int s에 대한 pointer이다

#### 괄호는? 괄호가 없으면 우선 순위: [ ] > \* 순서로

##### [ ] 는 \* 보다 우선순위가 높다

#### array of two int s를 가리키는 하나의 pointer.

## 10.7.1.2 Array of two pointers: 포인터 배열?

int \* pax[2]; // pax는 array of two pointers-to-int

### pax는 array이며 각 element는 pointers

<<Listing 10.16>>

zippo[m][n] == \*(\*(zippo + m) + n)

pz[m][n] == \*(\*(pz + m) + n)

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

int (\*pz)[2];

pz = zippo;

printf(" pz = %p, pz + 1 = %p\n",

pz, pz + 1);

printf("pz[0] = %p, pz[0] + 1 = %p\n",

pz[0], pz[0] + 1);

printf(" \*pz = %p, \*pz + 1 = %p\n",

\*pz, \*pz + 1);

printf("pz[0][0] = %d\n", pz[0][0]);

printf(" \*pz[0] = %d\n", \*pz[0]);

printf(" \*\*pz = %d\n", \*\*pz);

printf(" pz[2][1] = %d\n", pz[2][1]);

printf("\*(\*(pz+2) + 1) = %d\n", \*(\*(pz+2) + 1));

## 10.7.2 Pointer Compatibility

### 한 pointer를 다른 pointer 에 assign 하는 규칙: 타입 변환보다 더 엄격한 규칙 적용

int n = 5; double x;

int \* p1 = &n;

double \* pd = &x;

x = n; // implicit type conversion

pd = p1; // compile-time error

int \* pt;

int (\*pa)[3];

int ar1[2][3];

int ar2[3][2];

int \*\*p2; // a pointer to a pointer

* pointer compatibility?

pt = &ar1[0][0]; // both pointer-to-int

pt = ar1[0]; // both pointer-to-int

pt = ar1; // not valid – 이해할 수 있어야 한다

pa = ar1; // both pointer-to-int[3] – 이해할 수 있어야 한다

pa = ar2; // not valid

p2 = &pt; // both pointer-to-int \*

\*p2 = ar2[0]; // both pointer-to-int \*

p2 = ar2; // not valid

### nonvalid assignments : two pointers가 같은 type을 point하지 않으면 오류

#### pt = ar1; //error

##### pt는 single int를 가리킨다, ar1는 array of three int s를 가리킨다.

#### pa = ar1; pa = ar2;

##### pa는 array of three int를 가리킨다, ar1과 compatible하다

##### pa는 ar2와 compatible하지 않다, ar2는 an array of two int s를 가리킨다.

### 이중 pointer p2는 pointer-to-pointer-to-int이다, ar2는 pointer-to-array-of-two- int s이다 (pointer-to- int[2] ).

#### p2와 ar2는 다른 type이다

##### p2 = ar2; //not valid

#### \*p2는 type pointer-to- int이다

##### \*p2 = ar2[0]; //valid

##### ar2[0]는 type pointer-to- int에 대한 pointer

int x = 20;

const int y = 23;

int \* p1 = &x;

const int \* p2 = &y;

const int \*\* pp2;

p1 = p2; // not safe -- assigning const to non-const

p2 = p1; // valid -- assigning non-const to const

pp2 = &p1; // not safe -- assigning nested pointer types

* non-const pointer가 const pointer를 갖는 것: not safe
* const pointer가 non-const pointer를 갖는 것: okay

p2 = p1; // valid -- assigning non-const to const

pp2 = &p1; //pointer 에 대한 assign이 two levels of indirection으로 적용시에는 더 이상 safe 하지 않다

const int \*\*pp2;

int \*p1;

const int n = 13;

pp2 = &p1; // 허용되나 const qualifier가 무시됨?

\*pp2 = &n; // valid, both const, 그러나 p1이 n을 point 하게 됨

\*p1 = 10; // valid, 그러나 const n을 변경하는 효과가 발생하는 문제가 있음

## 10.7.2.1 C const and C++ const

const int y = 10;

const int \* p2 = &y;

int \* p1;

p1 = p2; // error in C++, possible warning in C

### C++에서 non-const pointer가 const pointer를 가질 수 없음

## 10.7.3 Functions와 다차원 Arrays

### 2차원 arrays을 처리하는 functions 작성

### 2차원 array의 각 행에 대하여 1차원 array를 처리하는 function을 작성하기

int junk[3][4] = { {2,4,5,8}, {3,5,6,9}, {12,10,8,6} };

int i, j;

int total = 0;

for (i = 0; i < 3 ; i++)

total += sum(junk[i], 4); // junk[i] -- one-dimensional array

### Junk는 pointer to an array of four int s.

void somefunction( int (\* pt)[4] );

void somefunction( int pt[][4] );

//Listing 10.17 The array2d.c Program

// array2d.c -- functions for 2d arrays

#include <stdio.h>

#include <stdlib.h>

#define ROWS 3

#define COLS 4

void sum\_rows(int ar[][COLS], int rows);

void sum\_cols(int[][COLS], int); // ok to omit names

int sum2d(int(\*ar)[COLS], int rows); // another syntax

int main(void)

{

int junk[ROWS][COLS] = {

{ 2,4,6,8 },

{ 3,5,7,9 },

{ 12,10,8,6 }

};

sum\_rows(junk, ROWS);

sum\_cols(junk, ROWS);

printf("Sum of all elements = %d\n", sum2d(junk, ROWS));

system("pause");

return 0;

}

void sum\_rows(int ar[][COLS], int rows)

{

int r;

int c;

int tot;

ar++;//rows의 증가, ar은 pointer로서 첫번째 index로 사용

for (r = 0; r < rows; r++)

{

tot = 0;

for (c = 0; c < COLS; c++)

{

//tot += (ar)[r][c]; printf("ar = %d\n", (ar)[r][c]);

tot += \*(\*(ar + r) + c);

printf("ar = %d\n", \*(\*(ar + r) + c));

}

printf("row %d: sum = %d\n", r, tot);

}

}

void sum\_cols(int ar[][COLS], int rows)

{

int r;

int c;

int tot;

for (c = 0; c < COLS; c++)

{

tot = 0;

for (r = 0; r < rows; r++)

tot += ar[r][c];

printf("col %d: sum = %d\n", c, tot);

}

}

int sum2d(int ar[][COLS], int rows)

{

int r;

int c;

int tot = 0;

//ar++;

for (r = 0; r < rows; r++) /// loop through rows

{

for (c = 0; c < COLS; c++) //// loop through columns

tot += ar[r][c];

}

return tot;

}

### Function은 ar을 an array of arrays of four int s로 간주

#### a과 junk는 same type: pointer-to-array-of-four- int s.

int sum2(int ar[][], int rows); // faulty declaration

### ar[1]은 ar+1과 같다. ar이 가리키는 object의 size를 알아야 한다.

int sum2(int ar[][4], int rows); // valid declaration

### ar+1은 “add 16 bytes to the address.”

#### [][] 표현으로는 function에서 pointer의 객체 타입을 알수가 없다

int sum2(int ar[3][4], int rows); // valid declaration, 3 ignored

typedef int arr4[4]; // arr4 array of 4 int typedef arr4 arr3x4[3]; // arr3x4 array of 3 arr4 int sum2(arr3x4 ar, int rows); // same as next declaration int sum2(int ar[3][4], int rows); // same as next declaration int sum2(int ar[][4], int rows); // standard form

### N 차원의 array를 사용하는 pointer 사용

int sum4d(int ar[][12][20][30], int rows);

int sum4d(int (\*ar)[12][20][30], int rows); // ar a pointer

//Listing 10.18 The Matrix Program

#include <stdio.h>

#include <stdlib.h>

#define ROWS 3

#define COLS 4

#define COLSD 5

#define COLSE 5

void transpose(const int a[][COLS], int rows, int b[][ROWS], int cols);

void product(const int a[][COLS], int rows, const int d[][COLSD], int rowsd, int e[][COLSE], int rowse);

void show(int\* a, int rows, int cols);

int main(void)

{

int a[ROWS][COLS] = { { 2, 4, 6, 8}, { 3, 5, 7, 9}, {12,10, 8, 6} };

printf("Showing Matrix a:\n");

show((int\*)a, ROWS, COLS);

int b[COLS][ROWS];

int d[COLS][5] = { { 1,2,3,4,5 },{ 1,1,1,1,1 },{ 2,2,2,2,2 },{ 3,3,3,3,3 } };

int e[ROWS][5];

transpose(a, ROWS, b, COLS);

printf("Transpose of a : ");

show((int\*)b, COLS, ROWS);

printf("Matrix d : ");

show((int\*)d, COLS, COLSD);

product(a, ROWS, d, COLS, e, ROWS);

printf("Product of MATRIX a and MATRIX d : ");

show((int\*)e, ROWS, COLSE);

system("pause");

return 0;

}

void transpose(const int a[][COLS], int rows, int b[][ROWS], int cols)

{

for (int i = 0; i < rows; i++)

{

for (int j = 0; j < cols; j++)

{

b[j][i] = a[i][j];

}

}

}

void product(const int a[][COLS], int rows, const int d[][COLSD], int rowsd, int e[][COLSE], int rowse)

{

int sum = 0;

for (int i = 0; i < rows; i++)

{

for (int j = 0; j < COLSD; j++)

{

for (int k = 0; k < rowsd; k++)

{

sum = sum + a[i][k] \* d[k][j];

}

e[i][j] = sum;

sum = 0;

}

}

}

void show(int\* a, int m, int n)

{

printf("\n{");

int i, j;

for (i = 0; i < m; i++)

{

printf("\n {");

for (j = 0; j < n; j++)

{

printf("%d ,", \*((a + i \* n) + j));

}

printf("} ,");

}

printf("\n}\n");

}

