8. Arrays

[ECE10002] C Programming

Agenda

- Introduction
- Arrays
- Passing Array as Function Arguments
- Multi-Dimensional Arrays
- Sorting

Introduction

- Derived types
  - Array: chapter 8
    - Collection of homogenous entries
  - Pointer: chapter 9
    - Variable to store address of variables
  - Structure/Union: chapter 12
    - Collection of heterogeneous entries
  - Enumeration: chapter 12
    - Finite list of identifiers

Introduction

- Motivation: Sometimes, we need to store and use a series of values of same data type.

  Ex) scores of 10 students

  - Representation using singleton variables

    int score0, score1, score2, … score9;

    - Not efficient to maintain many variables not related.

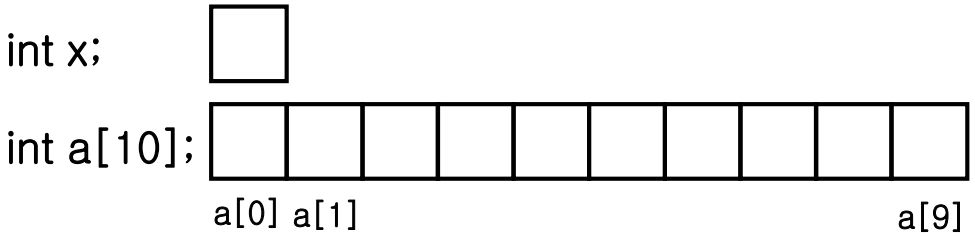
    Ex) reading 10 scores

```
printf("Input score of student 0: ");
scanf("%d", &score0);
printf("Input score of student 1: ");
scanf("%d", &score1);
...
printf("Input score of student 9: ");
scanf("%d", &score9);
```

Array

- Array: a series of data elements, usually of the same size and data type
  - Syntax: type arrayName[arraySize]
    - arraySize should be a constant!

```
Ex) int x;          // declaration of a variable
    int a[10];      // declaration of array of size 10
                    //         elements: a[0], a[1], …, a[9]
```



Array

- Accessing elements in array
  - Each element is a variable, accessed by its index (position in the array)
  - Syntax: arrayName[index] // []: index operator
    - Range of array index in C language: [0, size)

  - Integer variables can be used as array indices

Ex)

```
int a[10];          // array
int i = 0;          // counter variable
for(i = 0; i < 10; i++){
   a[i] = i * i;                          // square of i
   printf("a[%d] = %d₩n", i, a[i]);
}
```

Why Array?

- Why array?
  - Membership of elements is explicitly represented
    - int score[10]; // 10 variables to store scores
  - Efficient in manipulation

    Ex) reading 10 scores

```
int i = 0;
for(i = 0; i < 10; i++){
   printf("Input score of student %d: ", i);
   scanf("%d", &score[i]);
}
```

  - Appropriate to represent **list**, **vector**, **matrix**(2D array), etc.

Array Initialization

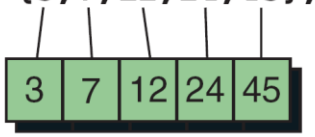
- Array initialization
  - Syntax: type arrayName[arraySize] = { e_0, e_1, ... };
    - If initial values are provided, array size can be omitted
    - An array can be partially initialized

(a) Basic Initialization

```
int numbers[5] = {3,7,12,24,45};
```

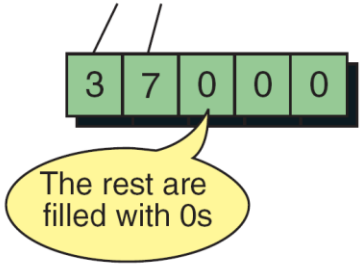
(b) Initialization without Size

```
int numbers[ ] = {3,7,12,24,45};
```



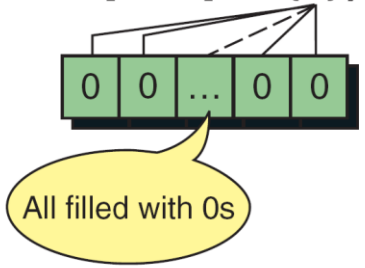
(c) Partial Initialization

```
int numbers[5] = {3,7};
```



(d) Initialization to All Zeros

```
int lotsOfNumbers [1000] = {0};
```

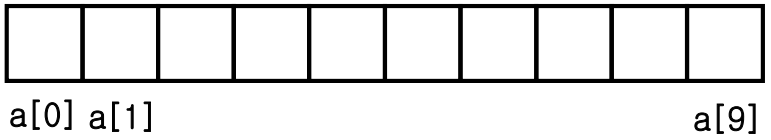


Examples

- Printing values

```
#define ArraySize 10          // use defined constants
int a[ArraySize];             // for array size, not a variable

int i = 0;
...
for(i = 0; i < ArraySize; i++)
    printf("a[%d] = %d₩n", i, a[i]);
```



- Exchanging elements (ex: exchanging a[1] and a[3])

```
int temp = a[1];
a[1] = a[3];
a[3] = temp;
```

Constants for Array Size

- Defined constant

  ```
  #define ArraySize 10
  int a[ArraySize];
  ```

  - Valid in Ansi-C (old standard), C99 (the most recent standard)
  - Available on all compilers

- Memory constant

  ```
  const int arraySize = 10;
  int a[arraySize];
  ```

  - Valid in C99, not in Ansi-C
  - Supported by **many** compilers.

- Variable

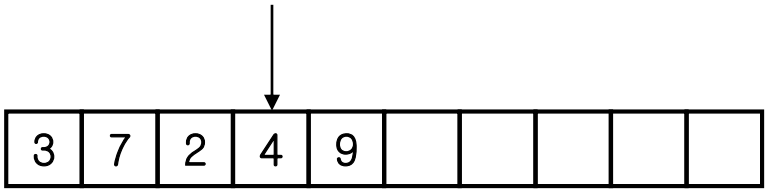
  ```
  int arraySize = 10;
  int a[arraySize];
  ```

  - Valid in C99, not in Ansi-C
  - Supported by **some** compilers.

Examples

- Search: Finding index of a value

```
for(i = 0; i < ArraySize; i++){
   if(a[i] == target)
      break;
}
// if i == ArraySize, it indicates target does not exist in a
```



Finding 4

Index Range Checking

- Index range of an array of size N is from 0 to N-1
  - C compiler does not check the boundary of an array
- Using invalid index causes unpredictable result.
  - Crash
  - Accessing garbage value
  - Modifying other variable

Ex) int a[10];

```
printf("a[10] = %d₩n", a[10]);
printf("a[-1] = %d₩n", a[-1]);
```

Index Range Checking

- Typical error patterns

```
int i, a[10];
```

  - Printing array

```
for(i = 0; i <= 10; i++)
  printf("a[%d] = %d₩n", i, a[i]);
```

  - Printing array in reverse order

```
for(i = 10; i >= 0; i--)
  printf("a[%d] = %d₩n", i, a[i]);

for(i = 0; i > 0; i--)
  printf("a[%d] = %d₩n", i, a[i]);
```

Precedence and Associativity

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Exercises

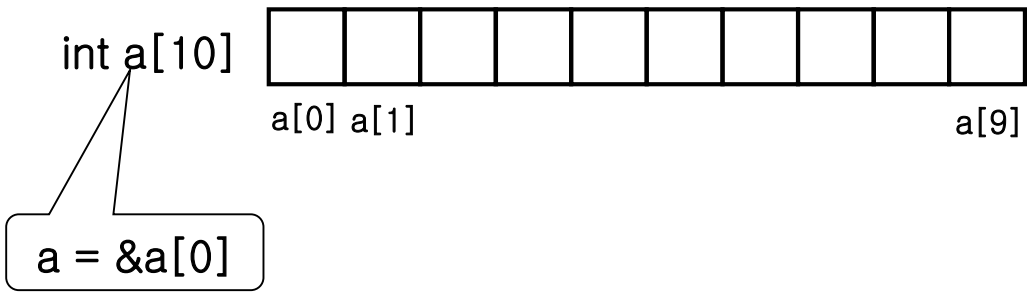
- Filling an array with random numbers
- Computing sum, mean, and variance
- Finding minimum and maximum

Array and Pointer

- Array name is a primary expression whose values is the address of the first element

Ex)

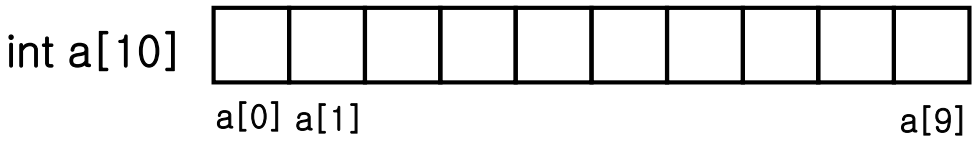
```
int a[10];
printf("a = %p, &a[0] = %p₩n", a, &a[0]);
printf("(a == &a[0]) = %d₩n", a == &a[0]);
```



Array and Pointer

- Index operator vs. pointer addition

```
int a[10];              // array declaration
    // a    = &a[0];    // *a = a[0]
    // a+1 = &a[1];    // *(a+1) = a[1]
    // a+2 = &a[2];    // *(a+2) = a[2]
    ...
    // a+9 = &a[9];    // *(a+9) = a[9]
```



Ex) // see ArrayAndPointer.c

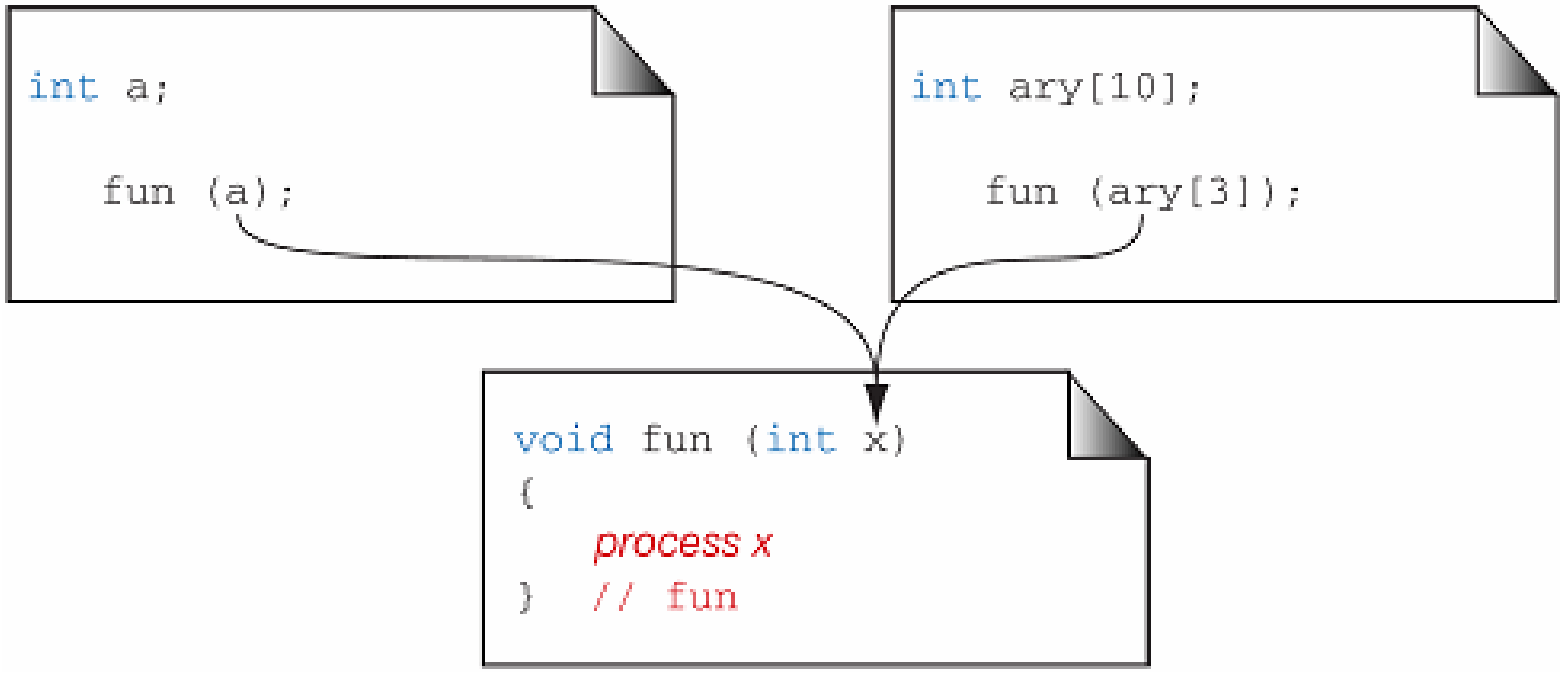
```
for(i = 0; i < 10; i++) {
   printf("a + %d = %p₩t", i, a+i);
   printf("*(a+%d) = %d, a[%d] = %d₩n", i, *(a+i), i, a[i]);
}
```

Agenda

- Introduction
- Arrays
- Passing Array as Function Arguments
- Multi-Dimensional Arrays
- Sorting

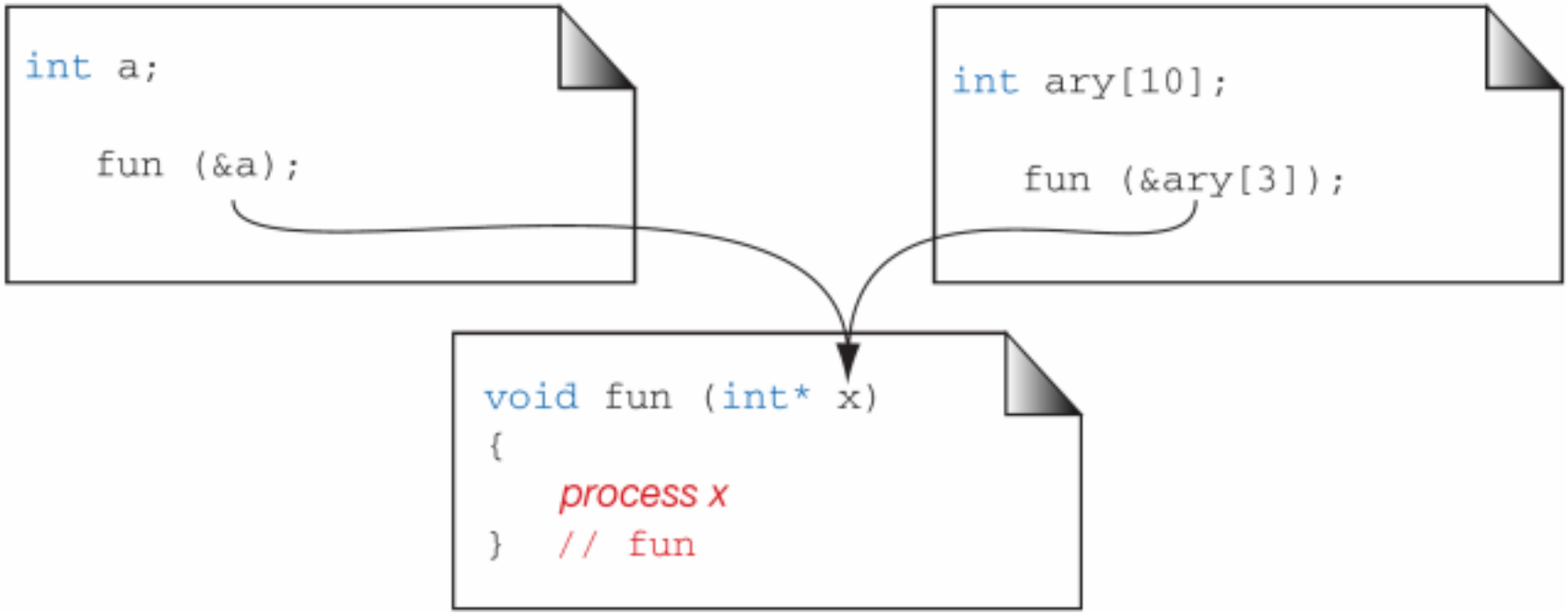
Passing Individual Elements

- Passing individual elements
  → Same as singleton variables



Passing Individual Elements

- Passing addresses
  → Same as singleton variables



Example

```
#include <stdio.h>

void Exchange(int *x, int *y);

int main()
{
    int a[10];
    int i = 0;

    for(i = 0; i < 10; i++)
      a[i] = i * 10;

    Exchange(&a[0], &a[3]);

    for(i = 0; i < 10; i++)
      printf("a[%d] = %d₩n", i, a[i]);
}
```

```
void Exchange(int *x, int *y)
{
    int hold = *x;
    *x = *y;
    *y = hold;
}
```

Passing Whole Array

- Note! In C language, array cannot be assigned

```
int a[10];
int b[10];
a = b;          // not allowed
```

- Passing array

Actual parameter

```
int array[10];
fun(array);
```

Formal parameter

```
void fun(int a[]);      or
void fun(int *a);
```

Example

```
#include <stdio.h>
double average(int array[]);

int main()
{
   double avg = 0.;
   int base[5] = {3, 7, 2, 4, 5};

   avg = average(base);
   printf("ave = %f₩n", avg);

   return 0;
}
```

```
double average(int array[])
{
   int sum = 0;
   int i = 0;

   for(i = 0; i < 5; i++)
      sum += array[i];

   return (sum / 5.);
}
```

Review on Arrays and Pointers

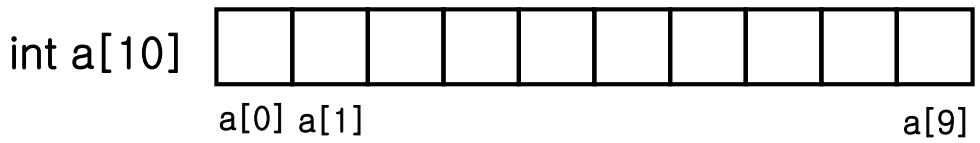
```
int a[10];                    // array declaration
```

- For any i,

```
a + i ==&a[i]
*(a + i) ≡ *&a[i]             // * and & cancels each other
```

- Therefore,

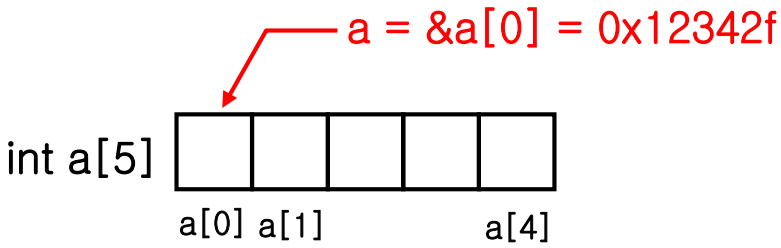
```
*(a + i) ≡ a[i]               // definition of the [] operator
```



Passing Whole Array

- In function call, C does not copy whole array, but passes the starting address

```
int main()
{
    int a[5] = {3, 7, 2, 4, 5}; // a = 0x12342f
    func(a);
    printf("a[0] = %d₩n", a[0]);
}
```




```
void func(int array[])
{
    array[0] = 0;        // &array[0] = 0x12342f
}
```

Agenda

- Introduction
- Arrays
- Passing Array as Function Arguments
- Multi-Dimensional Arrays
- Sorting

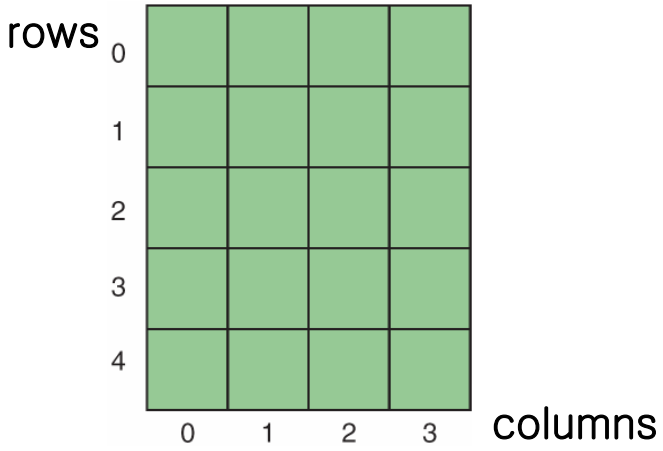
Multi-Dimensional Arrays

- One dimensional array

  Ex) int array1D[10];

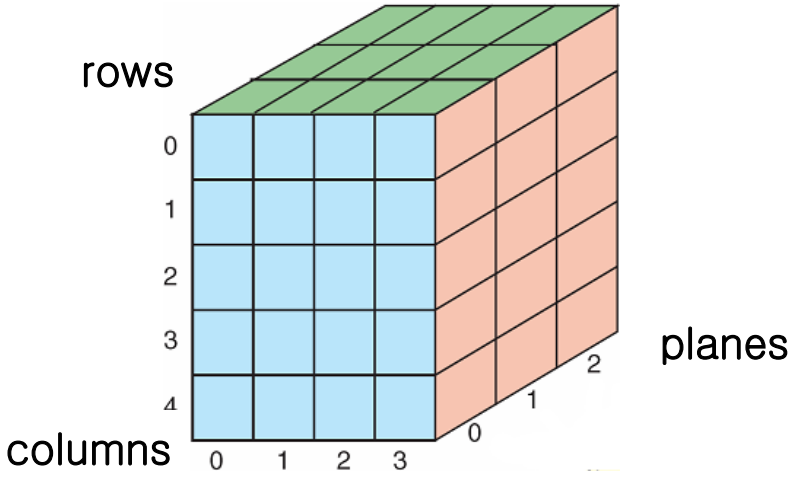
- Two dimensional array

  Ex) int array2D[5][4];



- Three dimensional array

  Ex) int array3D[3][5][4];



Declaration and Element Access

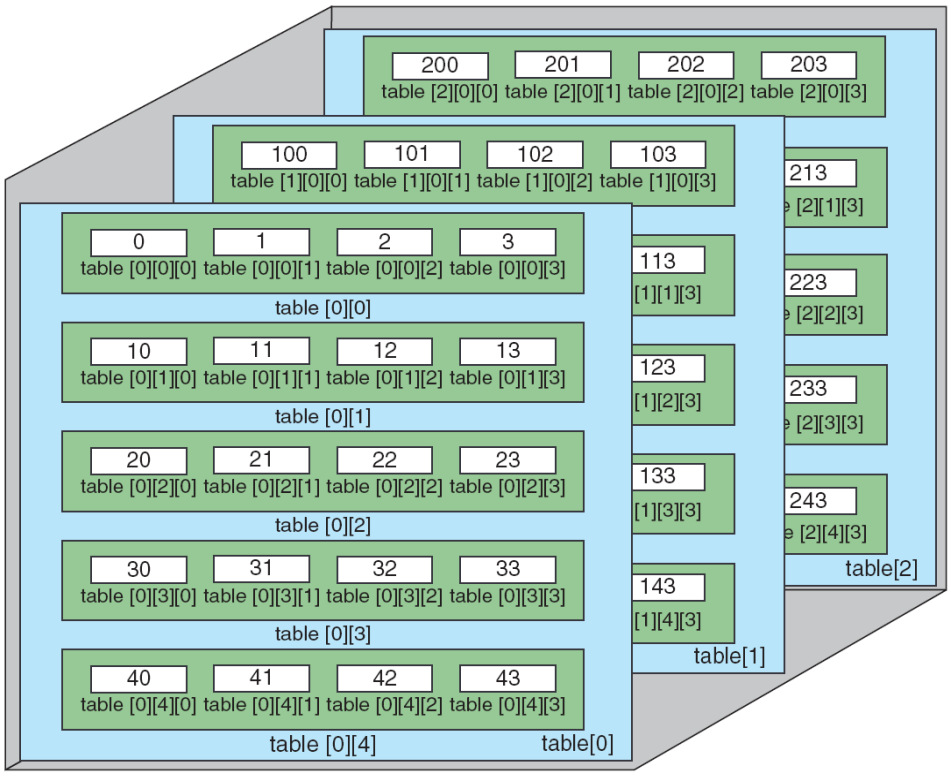
- Declaration
  - 2D array: type arrayName[$size_0$][$size_1$]
  - 3D array: type arrayName[$size_0$][$size_1$][$size_2$]
    …
  - N-D array: type arrayName[$size_0$][$size_1$]…[$size_{N-1}$]

- Element access
  - 2D array: arrayName[idx_0][idx_1]
  - 3D array: arrayName[idx0][idx1][idx2]
    …
  - N-D array: arrayName[idx0][idx1]…[idxN-1]

Multi-Dimensional Array

- D-dimensional array is an array of (D-1)-dimensional array.

  Ex) int table[3][5][4]



Example

- Array2D.c

```
#include <stdio.h>

int main()
{
    const int row = 5, col = 4;
    int a[row][col];
    int x = 0, y = 0;

    printf("a = %p₩n", a);
    for(y = 0; y < row; y++){
      printf("a[%d] = %p₩n", y, a[y]);
      for(x = 0; x < col; x++){
                printf("₩t&a[%d][%d] = %p₩n", y, x, &a[y][x]);
      }
      printf("₩n");
    }

    return 0;
}
```

Example: Execution result

- Execution result

```
a = 0022FEE0
a[0] = 0022FEE0
    &a[0][0] = 0022FEE0
    &a[0][1] = 0022FEE4
    &a[0][2] = 0022FEE8
    &a[0][3] = 0022FEEC

a[1] = 0022FEF0
    &a[1][0] = 0022FEF0
    &a[1][1] = 0022FEF4
    &a[1][2] = 0022FEF8
    &a[1][3] = 0022FEFC

a[2] = 0022FF00
    &a[2][0] = 0022FF00
    &a[2][1] = 0022FF04
    &a[2][2] = 0022FF08
    &a[2][3] = 0022FF0C

a[3] = 0022FF10
    &a[3][0] = 0022FF10
    &a[3][1] = 0022FF14
    &a[3][2] = 0022FF18
    &a[3][3] = 0022FF1C

a[4] = 0022FF20
    &a[4][0] = 0022FF20
    &a[4][1] = 0022FF24
    &a[4][2] = 0022FF28
    &a[4][3] = 0022FF2C
```

Initialization

- 2D array

```
Ex) int array2D[3][4] = {
        { 0, 1, 2, 3 },
        { 4, 5, 6, 7 },
        { 8, 9, 10, 11 }
    };
```

- 3D array

```
Ex) int array3D[2][3][2] = {
        {                   // plane 0
          { 0, 1 },         // row 0
          { 2, 3 },         // row 1
          { 4, 5 },         // row 2
        },
        {                   // plane 1
          { 6, 7 },         // row 0
          { 8, 9 },         // row 1
          { 10, 11 },       // row 2
        }
    };
```

Passing Multi-Dimensional Arrays

- Actual parameter

```
#define MAX_ROWS 12
#define MAX_COLS 10
int array2D[MAX_ROWS][MAX_COLS];
...
func(array2D);                    // just pass the array name
```

- Formal parameter declaration
  - Size of all but the highest dimension should be specified.

```
void func(int table[][MAX_COLS]);
```

Review: Passing Whole Array

- Note! In C language, array cannot be assigned

```
int a[10];
int b[10];
a = b;          // not allowed
```

- Passing array

Actual parameter

```
int array[10];
fun(array);
```

Formal parameter

```
void fun(int a[]);      or
void fun(int *a);
```

Example

```
#include <stdio.h>
double average(i          );

int main()
{
   double avg = 0.;
   int base[5] = {3, 7, 2, 4, 5};

   avg = average(base);
   printf("ave = %f₩n", avg);

   return 0;
}
```

```
double average(          )
{
   int sum = 0;
   int i =0;

   for(i = 0; i < 5; i++)
      sum += array[i];

   return (sum / 5.);
}
```

Example

```
#include <stdio.h>
double average(          );

int main()
{
   double avg = 0.;
   int base[3][5] = {
      {3, 7, 2, 4, 5},
      {5, 2, 1, 0, 7},
      {1, 3, 5, 9, 3}
   };

   ave = average(base);
   printf("ave = %f₩n", avg);

   return 0;
}
```

```
double average(          )
{
   int sum = 0;
   int i = 0, j = 0;

   for(i = 0; i < 3; i++)
      for(j = 0; j < 5; j++)
         sum += array[i][j];

   return (sum / 15.);
}
```

Agenda

- Introduction
- Arrays
- Passing Array as Function Arguments
- Multi-Dimensional Arrays
- Sorting

Sorting

- Sorting: process of arranging items in some sequence or list

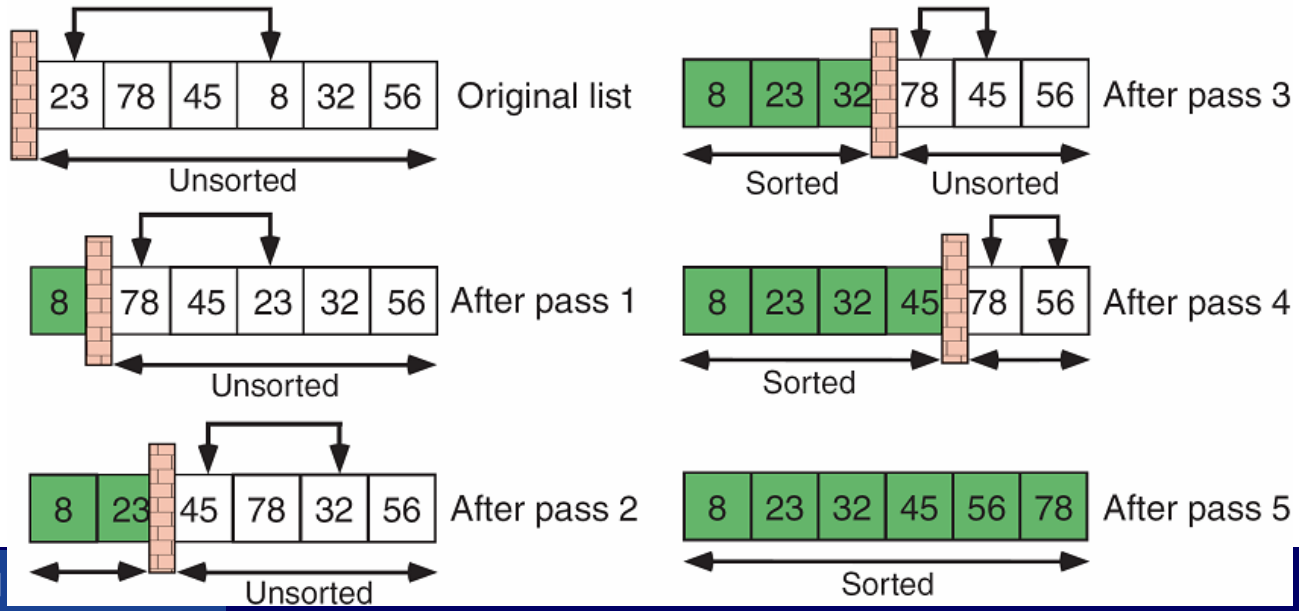
  Sequence: (4, 3, 8, 1, 9, 2)

  - Sorted in ascending order: (1, 2, 3, 4, 8, 9)
  - Sorted in descending order: (9, 8, 4, 3, 2, 1)

- Sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Quicksort/mergesort/heapsort/…
  - ETC.

Selection Sort

- Idea (sorting in ascending order)
  - List is divided into two sublists, **sorted** and **unsorted**
    - Initially, all elements are in **unsorted**
  - At each pass, select the smallest from **unsorted** sublist and put it at the end of **sorted** sublist
    - **sorted** gains one, but **unsorted** loses one.
  - Repeat n times



Selection Sort

```
void selectionSort(int list[], int size)
{
    int current = 0;                        // start of the unsorted list

    for(current = 0; current < size - 1; current++){
        int smallest = 0;                   // location of the smallest element
        int walk = 0;                       // variable to traverse the unsorted list
        int tempData = 0;                   // temporary variable for exchange

        // find the smallest in the unsorted list
        smallest = current;
        for(walk = current + 1; walk < size; walk++)
            if(list[walk] < list[smallest])
                smallest = walk;

        // exchange the smallest with the first of unsorted list
        tempData = list[current];
        list[current] = list[smallest];
        list[smallest] = tempData;
    }
}
```