
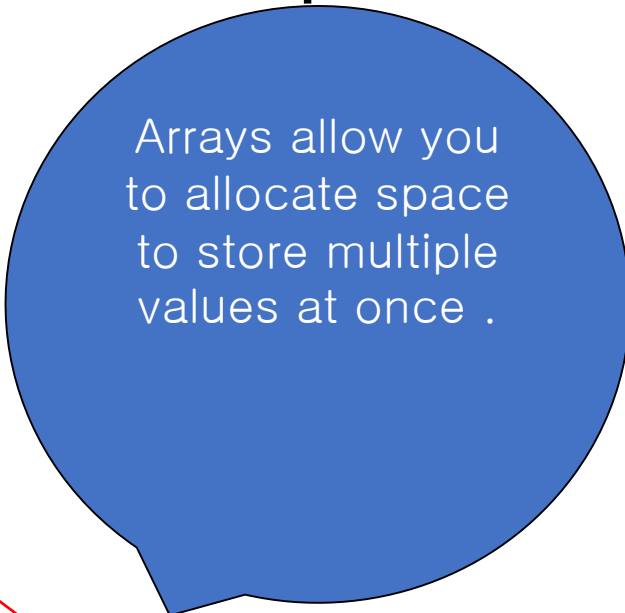


# Ch.10 Array

# What you will learn in this chapter

- 
- Understanding the concept of repetition
  - The concept of arrays
  - Declaration and initialization of arrays
  - One-dimensional array
  - Multidimensional array



Arrays allow you to allocate space to store multiple values at once .

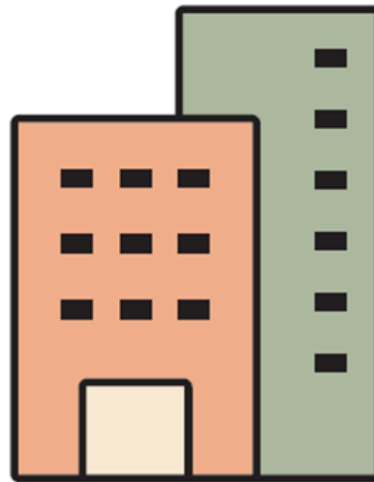


# Array

- Arrays allow you to create multiple variables at once .
- `int s[10];`



House(Variable)



Apartment(Array)

# The need for arrays

// Use regular variables

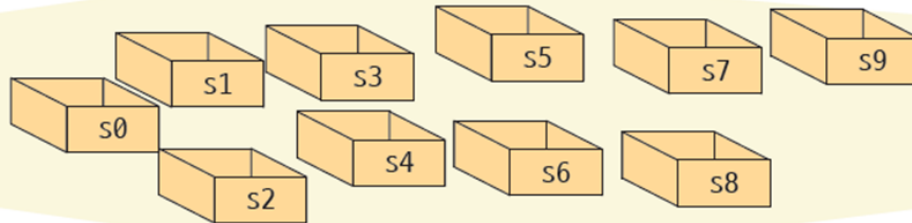
```
int s0;
```

```
int s1;
```

```
...
```

```
int s9;
```

It's hard to manipulate  
since it has a separate name



// use array

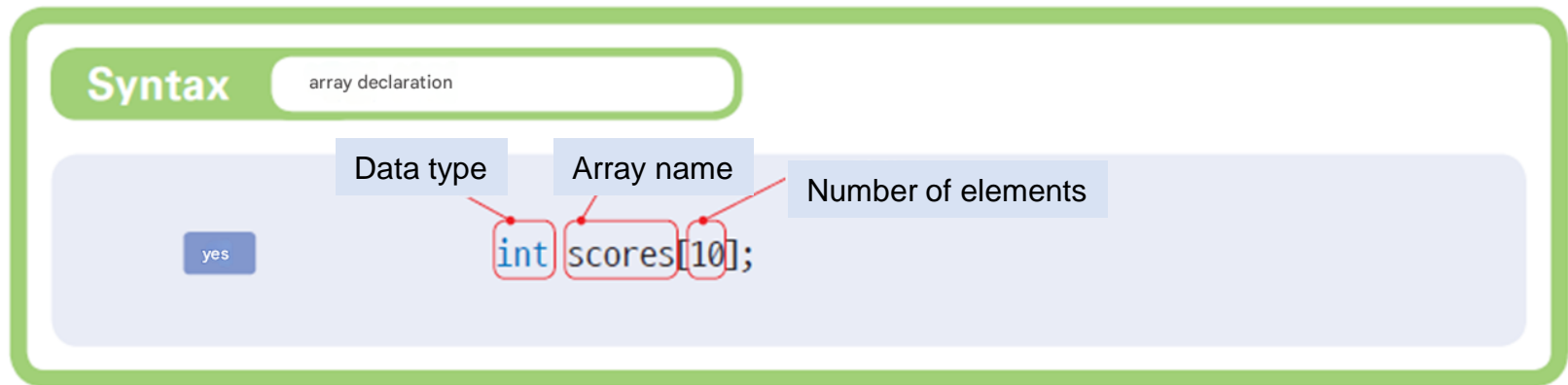
```
int s[10];
```



# Array of characteristic

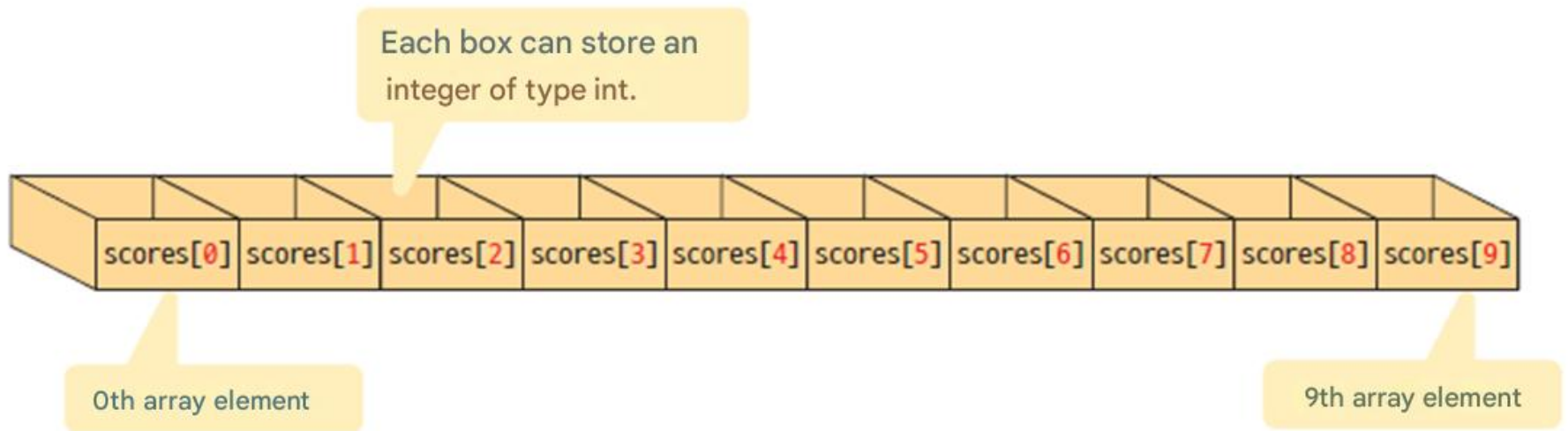
- Arrays are **stored in a contiguous space** in memory. For example, the array elements `s[0]` and `s[1]` above are physically adjacent to each other in memory.
- The biggest advantage of an array is that you can **access and process related data sequentially**. If the related data have different names, you would have to remember each name.
- However, if they share one name and only have different index numbers, they are very easy to remember and convenient to use.

# Array declaration



# Array elements and index

- *Index* : Number of array elements



# Example of array declaration

```
int score[60];
```

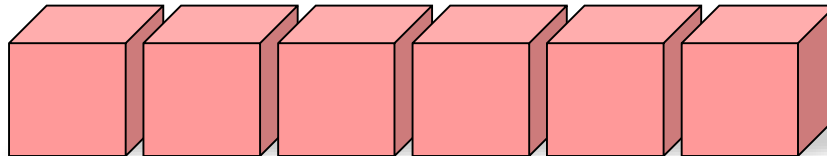
// Array score with 60 int type values

```
float cost[12];
```

// Array cost with 12 float type values

```
char name[50];
```

// Array name with 50 char type values





# caution

You should always **use constants** when representing the size of an array. Using variable as the size of an array will result in a compilation error. Also, if the size of the array is **negative, 0, or a real number, it is a compilation error.**

## warning

You should always use constants when representing the size of an array. Using a variable as the size of an array will result in a compilation error.

Also, if the size of the array is negative, 0, or a real number, it is a compilation error.

```
int scores[];           // Error! Array size must be specified
int scores[size];       // The size of an array cannot be a variable!
int scores[-2];         // The size of the array cannot be negative
int scores[6.7];        // The size of the array must not be a real number
```



# Using symbolic constants

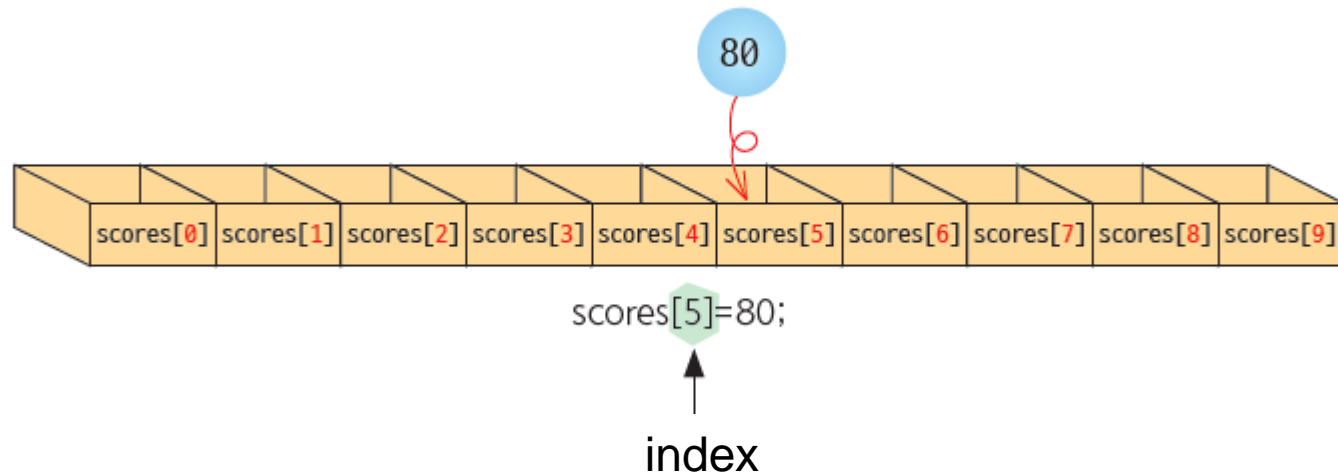
Usually, when declaring an array, the size of the array is specified as a symbolic constant created with the `#define`.

```
#define SIZE 10  
int scores[SIZE];
```

It becomes easy to change the size of the array.



# Accessing array elements

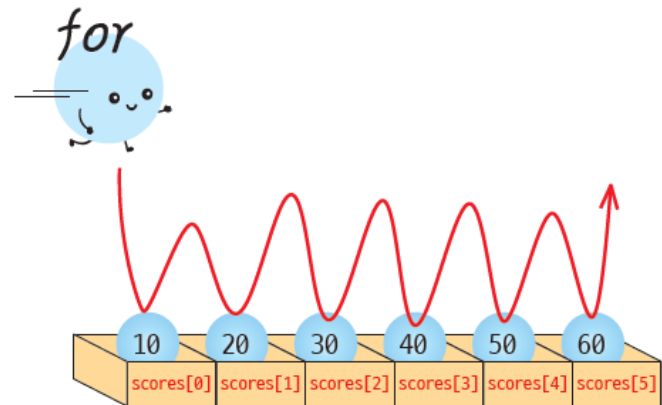


```
scores[5] = 80;  
scores[1] = scores[0];  
scores[ i ] = 100;      // i is an integer variable  
scores[i+2] = 100;      // The formula becomes the index .  
scores[index[3]] = 100; // index[] is an array of integers
```

# Array Basics Example

- Let's start with a basic example of declaring an array and assigning values to array elements. The for loop is useful when processing array elements one by one.

```
scores[0]=10  
scores[1]=20  
scores[2]=30  
scores[3]=40  
scores[4]=50
```



# Array Basics Example

```
#include <stdio.h>

int main( void )
{
    int i ;
    int scores[5];

    scores[0] = 10;
    scores[1] = 20;
    scores[2] = 30;
    scores[3] = 40;
    scores[4] = 50;

    for ( i =0; i < 5; i ++){
        printf ( "scores[%d]=%d\n" , i , scores[ i ] );
    }
    return 0;
}
```

```
scores[0]=10
scores[1]=20
scores[2]=30
scores[3]=40
scores[4]=50
```

# Array and Loop

- The biggest advantage of arrays is that you can easily process the elements of the array using a loop.




```
scores[0] = 0;  
scores[1] = 0;  
scores[2] = 0;  
scores[3] = 0;  
scores[4] = 0;
```

```
#define SIZE 5  
...  
for ( i = 0 ; i < SIZE ; i ++ )  
    scores[ i ] = 0;
```



# Fill array with random numbers

- This is an example of defining an array and using a repeating structure to initialize and print the values of the array elements with random numbers .



```
scores[0]=41  
scores[1]=67  
scores[2]=34  
scores[3]=0  
scores[4]=69
```

# Fill array with random numbers

```
#include <stdio.h>
#include <stdlib.h> // srand
#include <time.h>   // time
#define SIZE 5

int main( void )
{
    int i ;
    int scores[SIZE];

    srand ((unsigned)time(NULL));
    for (i = 0; i < SIZE; i++)
        scores[ i ] = rand() % 100;

    for (i = 0; i < SIZE; i++)
        printf( "scores[%d]=%d\n" , i, scores[i]);

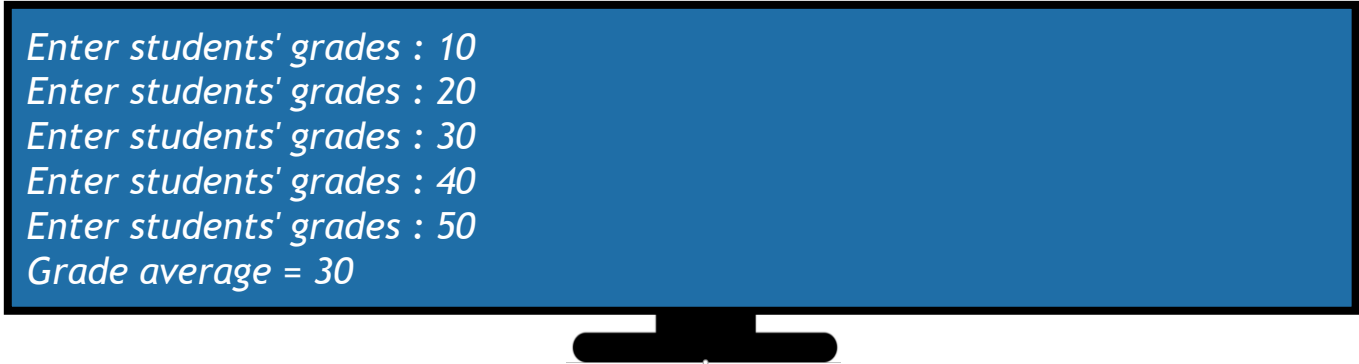
    return 0;
}
```

```
scores[0]=41
scores[1]=67
scores[2]=34
scores[3]=0
scores[4]=69
```



# Example #3: Calculating the grade average

- five students using arrays. Imagine how difficult it would have been if we had used five variables instead of an array .



```
Enter students' grades : 10
Enter students' grades : 20
Enter students' grades : 30
Enter students' grades : 40
Enter students' grades : 50
Grade average = 30
```

# Example #3: Calculating the grade average

```
#include <stdio.h>
#define STUDENTS 5
```

```
int main( void )
{
```

```
    int scores[ STUDENTS ];
    int sum = 0;
    int i , average;
```

```
    for (i = 0; i < STUDENTS ; i++) {
        printf ( " Enter students' grades : " );
        scanf ( "%d" , &scores[ i ]);
    }
```

```
    for (i = 0; i < STUDENTS ; i++) {
        sum += scores[ i ];
    }
```

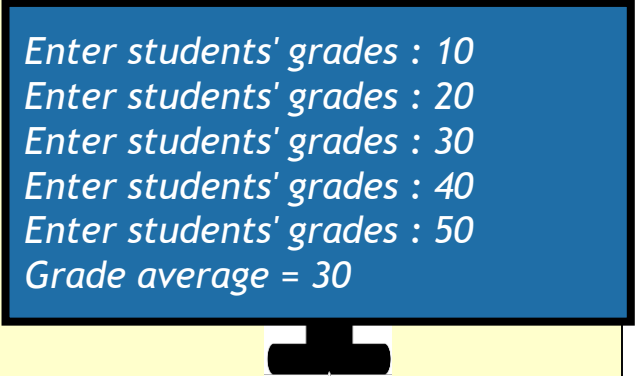
```
    average = sum / STUDENTS ;
    printf ( " Grade average = %d\n" , average);
```

```
    return 0;
```

```
}
```



Practice

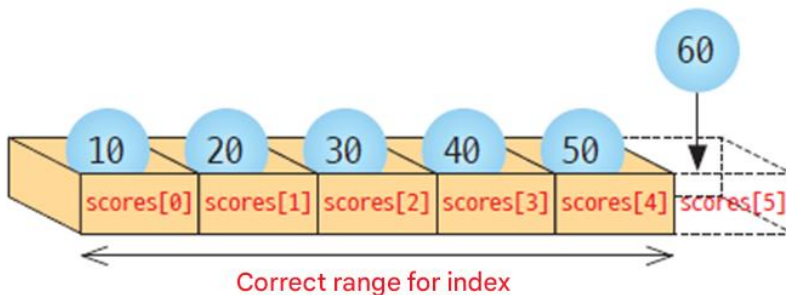


```
Enter students' grades : 10
Enter students' grades : 20
Enter students' grades : 30
Enter students' grades : 40
Enter students' grades : 50
Grade average = 30
```

# Bad index problem

- If the index exceeds the size of the array, the program will cause a fatal error .
- In C, it is the programmer's responsibility to ensure that the index is not out of range .

```
int scores[5];  
...  
scores[5] = 60; // Fatal error !
```

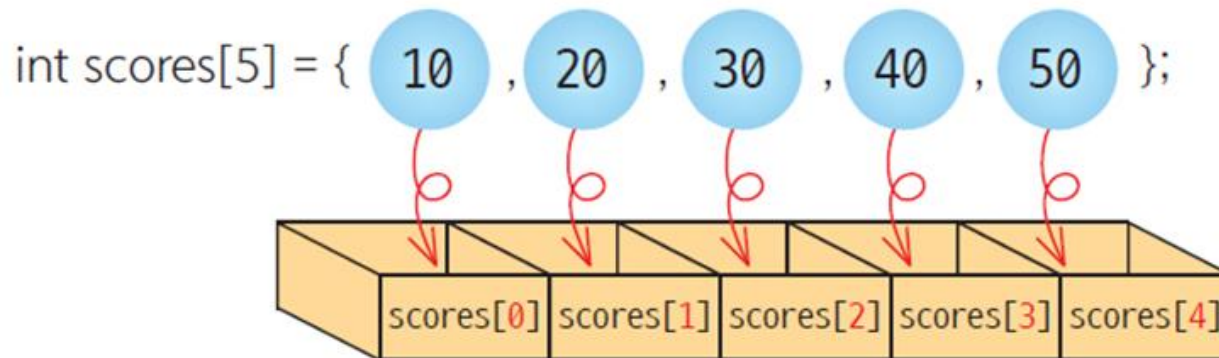


You should not store data where it does not exist.



# Initializing an array

- To initialize an array, separate the initial values with commas, enclose them in curly brackets { }, and assign them when declaring the array.



The initial values of the elements are listed within curly brackets, separated by commas.



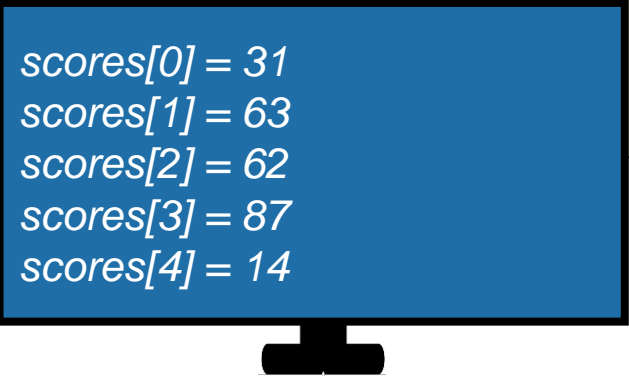
# Array initialization example

```
#include <stdio.h>
#define SIZE 5

int main( void )
{
    int i;
    int scores[SIZE] = { 31, 63, 62, 87, 14 };

    for (i = 0; i < SIZE; i++)
        printf( "scores[%d] = %d\n" , i, scores[i]);

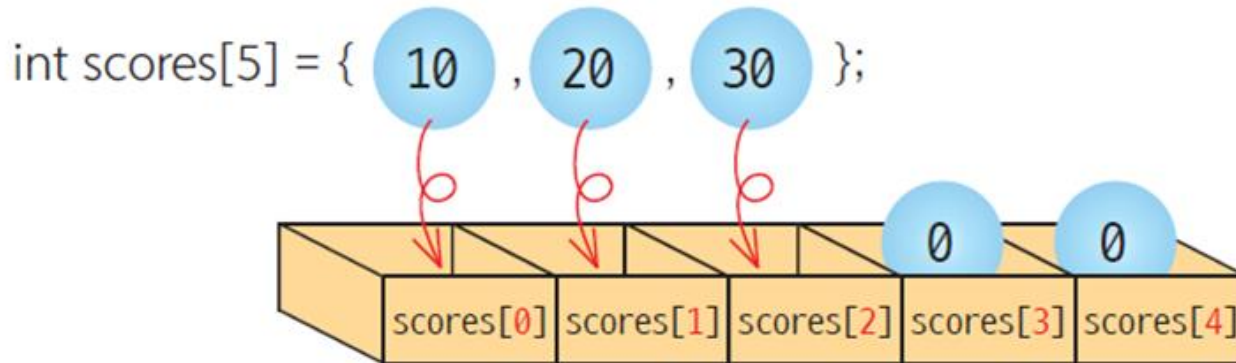
    return 0;
}
```



```
scores[0] = 31
scores[1] = 63
scores[2] = 62
scores[3] = 87
scores[4] = 14
```

# Initializing an array

- If the number of initial values is less than the number of elements, only the elements in front are initialized. All remaining array elements are initialized to 0.



If you give only some of the initial values, the remaining elements will be initialized to 0.



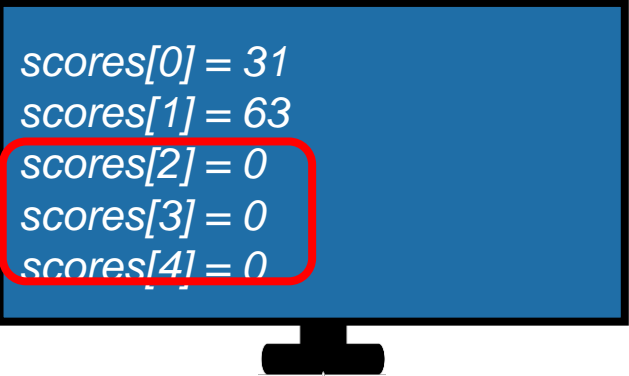
# Array initialization example

```
#include <stdio.h>
#define SIZE 5

int main( void )
{
    int i ;
    int scores[SIZE] = { 31, 63 };

    for (i = 0; i < SIZE; i++)
        printf( "scores[%d] = %d\n" , i, scores[i]);

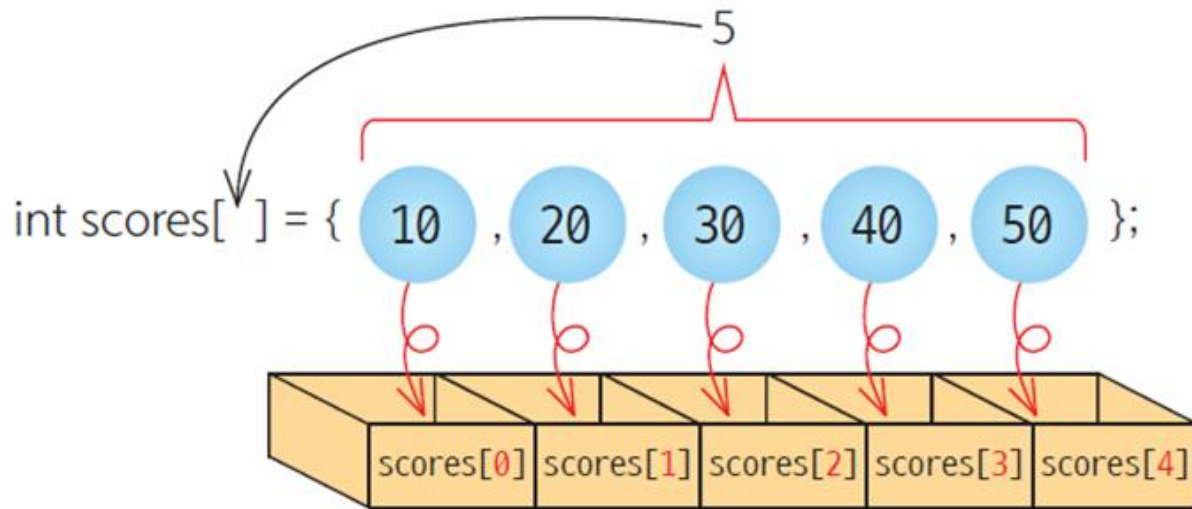
    return 0;
}
```



```
scores[0] = 31
scores[1] = 63
scores[2] = 0
scores[3] = 0
scores[4] = 0
```

# Initializing an array

- If you leave the array size empty, the compiler will automatically **set the array size to the number of initial values.**



If the size of the array is not given, the number of initial values becomes the size of the array.

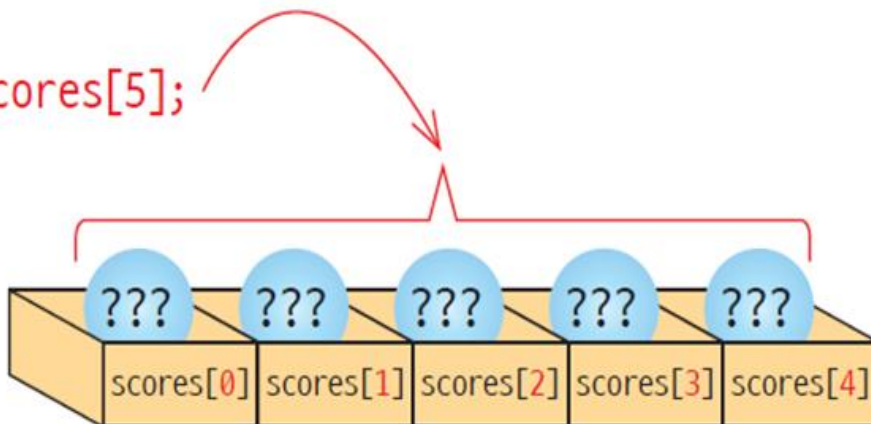




# What if no initial value was given ?

- If the array is declared as a local variable and no initial value is given, it has no meaning, just like a normal local variable. The garbage value is included .

```
int main(void)
{
    int scores[5];
    ...
}
```



If you declare an array as a local variable, the uninitialized array will contain garbage values.



# Array initialization example

```
#include <stdio.h>
#define SIZE 5

int main( void )
{
    int i ;
    int scores[SIZE] ;

    for (i = 0; i < SIZE; i++)
        printf( "scores[%d] = %d\n" , i, scores[i]);

    return 0;
}
```

```
scores[0]=4206620
scores[1]=0
scores[2]=4206636
scores[3]=2018779649
scores[4]=1
```

# warning



If you try to initialize all elements of an array to 10 as follows, It will be an error.

```
int scores[10] = { 10 };
```

At this time, only the first element becomes 10 and the remaining elements are all 0.

# reference

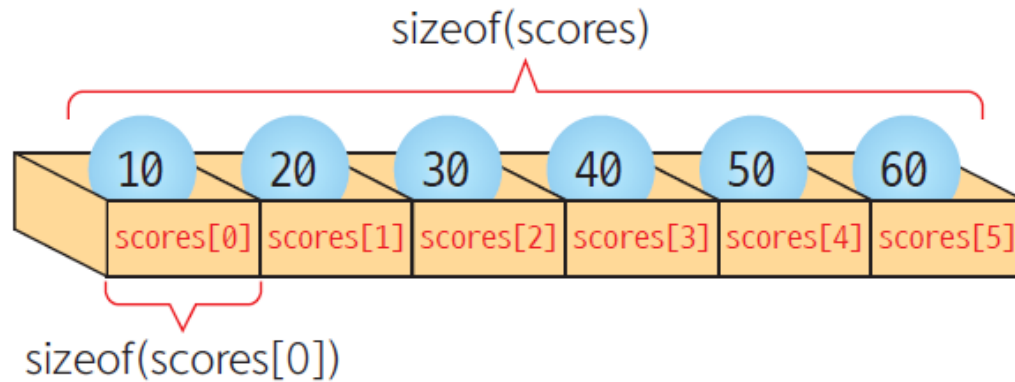


## Note

You cannot assign values by enclosing them in braces except when initialing an array. To save, you must assign a value to each array element.

```
#define SIZE 3
int main(void)
{
    int scores[SIZE];
    scores = { 6, 7, 8 }; // Compile error!!
}
```

# Counting the number of array elements



```
int scores[] = { 1, 2, 3, 4, 5, 6 };  
int i , size;
```

```
size = sizeof (scores) / sizeof (scores[0]);
```

Calculate the number  
of array elements

```
for ( i = 0; i < size ; i ++)  
    printf ( "%d " , scores[ i ] );
```

# Copying an array



```
int a[SIZE] = {1, 2, 3, 4, 5};  
int b[SIZE];  
a = b; // compile error !
```

Wrong way



```
int a[SIZE] = {1, 2, 3, 4, 5};  
int b[SIZE];  
int i ;
```

```
for ( i = 0; i < SIZE; i ++)  
    a[ i ] = b[ i ];
```

Copy each element  
one by one

The right  
way

# Comparing arrays

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

```
#define SIZE 5
```

```
int main( void )
```

```
{
```

```
    int i ;
```

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

```
    int b[SIZE] = { 1, 2, 3, 4, 5 };
```

```
    if ( a == b ) // ① Incorrect array comparison
```

```
        printf ( " Incorrect result .\n" );
```

```
    else
```

```
        printf ( " Incorrect result .\n" );
```



# Comparing arrays



```
for ( i = 0; i < SIZE ; i ++ ) // ② Correct array comparison
{
    if ( a[ i ] != b[ i ] )
    {
        printf ( "a[] and b[] are not equal .\n" );
        return 0;
    }
}
printf ( "a[] and b[] are equal .\n" );
return 0;
}
```

Compare  
elements  
**one by one**



# Lab: Finding the minimum






- When we buy products on the Internet, we search for the cheapest price through price comparison sites.
- similar to the problem of finding **the minimum value** among the integers in an array .



# Execution results

Practice

```
-----  
1  2  3  4  5  6  7  8  9 10  
-----  
28 81 60 83 67 10 66 97 37 94  
  
The minimum value is 10 .
```

Store	Customer rating	Inventory	Price	Total price
 Your Trusted Source since 1983	★★★★★ <a href="#">Rate this store</a> <a href="#">See store profile</a>	In stock Great Accessory Prices	Price: \$312.00 Tax: \$0.00 Shipping: Free	<b>\$312.00</b> Your best price <a href="#">Shop now</a>
	★★★★★ <a href="#">Rate this store</a> <a href="#">See store profile</a>	In stock	Price: \$312.95 Tax: \$0.00 Shipping: Free	<b>\$312.95</b> <a href="#">Shop now</a>
	★★★★☆ <a href="#">Rate this store</a> <a href="#">See store profile</a>	In stock	Price: \$312.95 Tax: \$0.00 Shipping: Free	<b>\$312.95</b> <a href="#">Shop now</a>
	★★★★☆ <a href="#">Rate this store</a> <a href="#">See store profile</a>	In stock	Price: \$313.00 Tax: \$0.00 Shipping: Free	<b>\$313.00</b> <a href="#">Shop now</a>
	Not yet rated <a href="#">Rate this store</a> <a href="#">See store profile</a>	In stock	Price: \$316.50 Tax: \$0.00 Shipping: Free	<b>\$316.50</b> <a href="#">Shop now</a>

# Algorithm

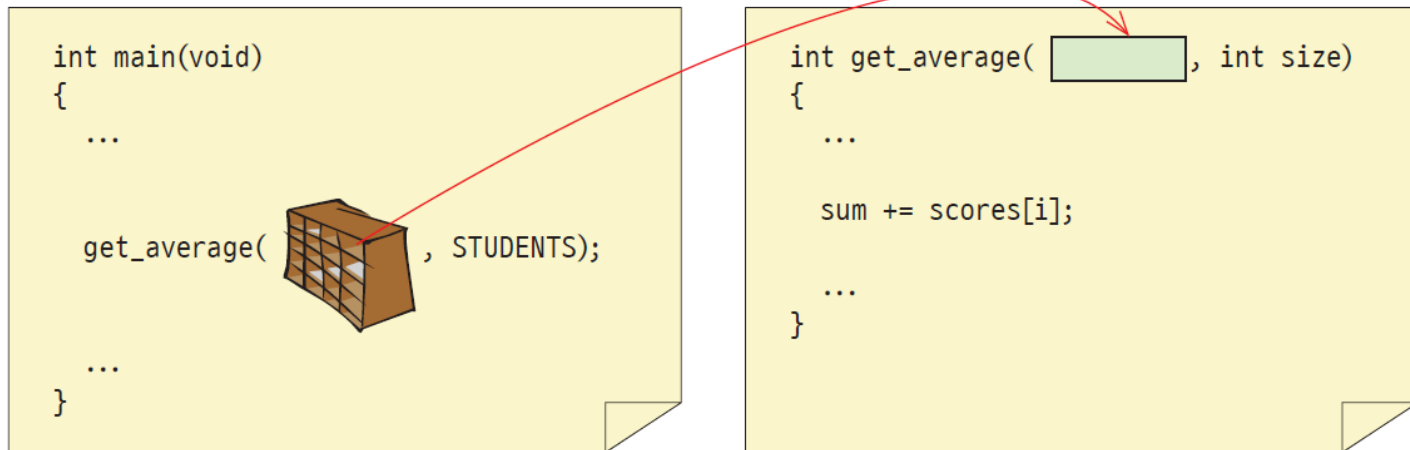
1. *Initialize the elements of array prices[] to 28 81 60 83 67 10 66 97 37 94.*
2. *assume that the first element is the minimum value minium.*
3. *for( i =1; i < size of array ; i ++)*
4.     *if ( prices[ i ] < minimum )*
5.         *minimum = prices[ i ]*
6. *When the iteration ends, the minimum value is stored in minimum .*

```
#include < stdio.h >
#define SIZE 10
int main( void )
{
    int prices[SIZE] = {28, 81, 60, 83, 67, 10, 66, 97, 37, 94 };
    int i , minimum;

    minimum = prices[0];
    for ( i = 1; i < SIZE; i ++ ) {
        if ( prices[ i ] < minimum ) {
            minimum = prices[ i ];
        }
    }
    printf ( " The minimum value is %d.\n" , minimum);
    return 0;
}
```

# Arrays and functions

- In the case of arrays, **the original is passed, not a copy.**



# Arrays and functions

- `void func(int arr[], int size);`
  - Although it looks like an array, it is actually a pointer pointing to the first*`void func(int* arr, int size);`*

When passing an array, the size is usually also passed.

- `int arr[5];`  
`int* p = arr; // OK`  
  
`sizeof(arr); // total size: 20 (5 * 4)`  
`sizeof(p); // pointer size: 8 (64bit)`

# Arrays and functions

Practice

```
#include <stdio.h>
#define STUDENTS 5
int get_average ( int scores[], int n);

int main( void )
{
    int scores[STUDENTS] = { 1, 2, 3, 4, 5 };
    int avg ;

    avg = get_average (scores, STUDENTS);
    printf ( "The average is %d .\n" , avg );
    return 0;
}

int get_average ( int scores[], int n)
{
    int i ;
    int sum = 0;

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

If the argument is an array ,  
The address of the array is passed.  
Pass by reference (&scores[0])

The original of the array  
is passed to score[]  
score work like pointer\*

The average is 3 .

# When array is an argument of a function

```
#include <stdio.h>
#define SIZE 7
```


```
void modify_array ( int a[], int size);
void print_array ( int a[], int size);
```

```
int main( void )
{
    int list[SIZE] = { 1, 2, 3, 4, 5, 6, 7 };

    print_array ( list , SIZE);
    modify_array ( list , SIZE);
    print_array ( list , SIZE);

    return 0;
}
```

Arrays are passed by address .





# When array is an argument of a function


- Can modify values

```
void modify_array ( int a[], int size)
{
    int i ;

    for ( i = 0; i < size; i ++)
        ++a[ i ];
}

void print_array ( int a[], int size)
{
    int i ;

    for ( i = 0; i < size; i ++)
        printf ( "%3d " , a[ i ]);
    printf ( "\n" );
}
```



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

# How to prevent changes to the original array

```
void print_array ( const int a[], int size)
{
    ...
    a[0] = 100; // Compile error !
}
```

a[] cannot be changed inside the function



The const keyword means that it cannot be changed, right ?



const means that it cannot be changed .

# What is sorting?

- Sorting is arranging items in ascending or descending order of size.
- Sorting is one of the most basic and important algorithms in computer engineering.



# What is sorting ?

- Sorting is essential for data exploration .  
( Example ) What if the words are not sorted in the dictionary?



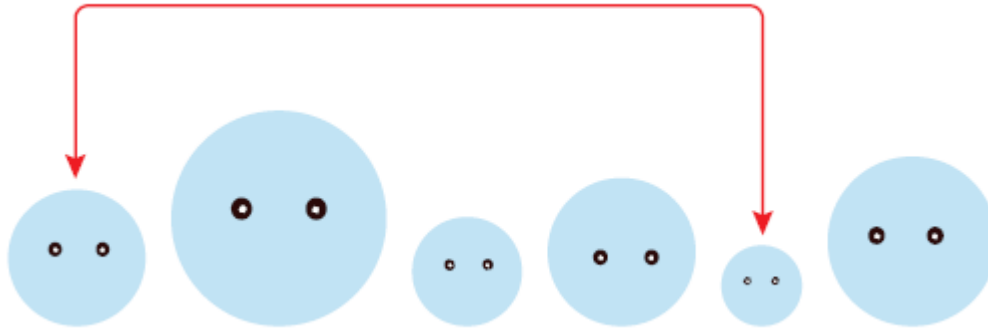
# Selection sort

- Selection sort : Select the minimum value from the unsorted numbers and exchange it with the first element of the array.

left arrangement	right arrangement	explanation
( )	(5,3,8,1,2,7)	initial state
(1)	(5,3,8,2,7)	1Choose
(1,2)	(5,3,8,7)	2Choose
(1,2,3)	(5,8,7)	3Choose
(1,2,3,5)	(8,7)	5Choose
(1,2,3,5,7)	(8)	7Choose
(1,2,3,5,7,8)	()	8Choose

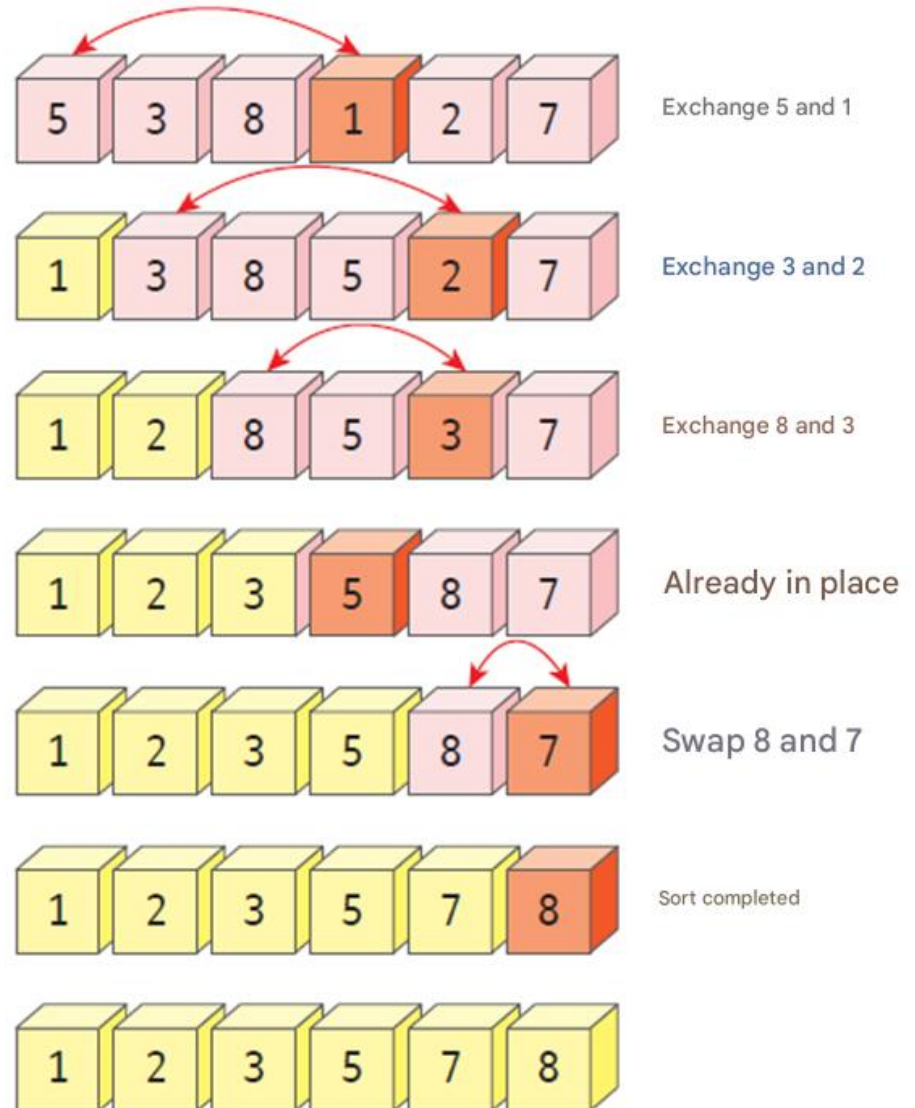
# Array If you only want to use one ?

- to handle the value in the first place, and we can do this by taking advantage of the fact that the place where the minimum value was is empty



# Selection sort

- Selection sort : Select the minimum value from the unsorted numbers and exchange it with the first element of the array.



# Selection Sort

```
#include <stdio.h>
#define SIZE 10

int main( void )
{
    int list[SIZE] = { 3, 2, 9, 7, 1, 4, 8, 0, 6, 5 };
    int i , j, temp, least;

    for ( i = 0; i < SIZE-1; i ++ )
    {
        least = i ;
        for ( j = i + 1; j < SIZE; j++ )
            if (list[j] < list[least])
                least = j;

        temp = list[ i ];
        list[ i ] = list[least];
        list[least] = temp;
    }
}
```

an inner for loop, it finds the minimum value from the (i+1) th element to the last element of the array. If a smaller integer is found by comparing it to the current minimum value, the index containing that integer is stored in least.

Swap list[ i ] and list[least]



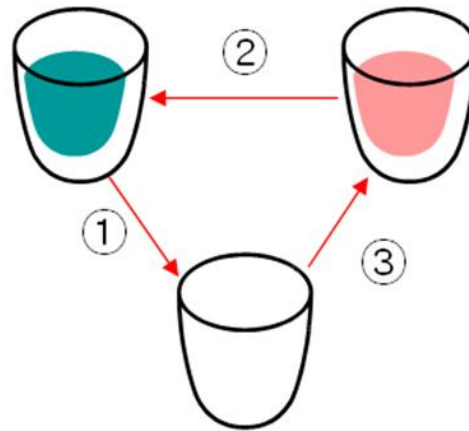
# Selection Sort

```
for ( i = 0; i < SIZE; i ++)  
    printf ( "%d " , list[ i ] );  
  
    printf ( "\n" );  
return 0;  
}
```

0 1 2 3 4 5 6 7 8 9

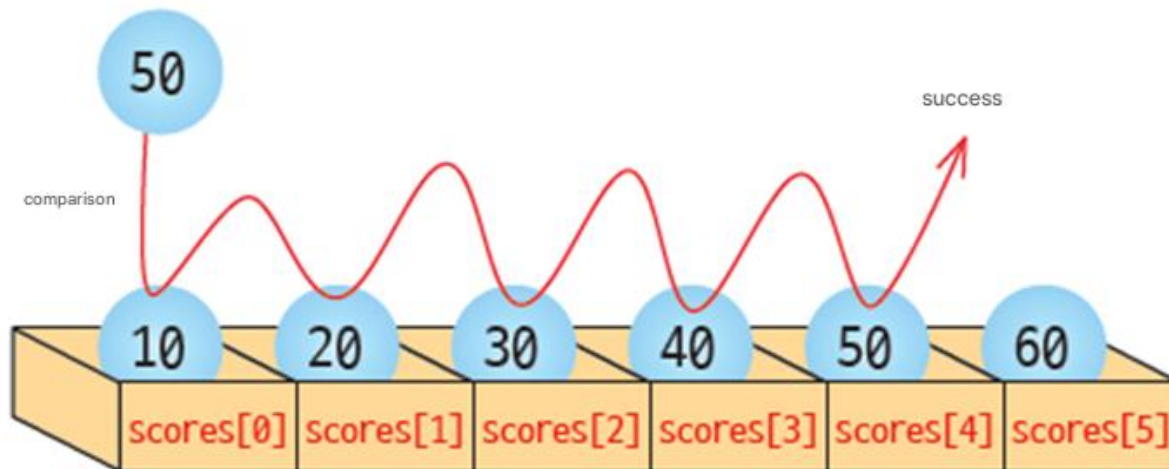
# When exchanging the values of variables

- Do not do the following:
  - `list[ i ] = list[least];` // The existing value of `list[ i ]` is destroyed !
  - `list[least] = list[ i ];`
- The right way
  - `temp = list[ i ];`
  - `list[ i ] = list[least];`
  - `list[least] = temp;`



# Sequential search

- Sequential The search is Array of The elements In order singly Take it out With the navigation keys By comparison desirous The value Go to method



# Sequential search

```
#include <stdio.h>
#define SIZE 10

int main( void )
{
    int key, i ;
    int list[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    printf ( " Enter the value to search for :");
    scanf ( "%d" , &key);

    for ( i = 0; i < SIZE; i ++ )
        if (list[ i ] == key)
            printf ( " Search success index = %d\n", i );

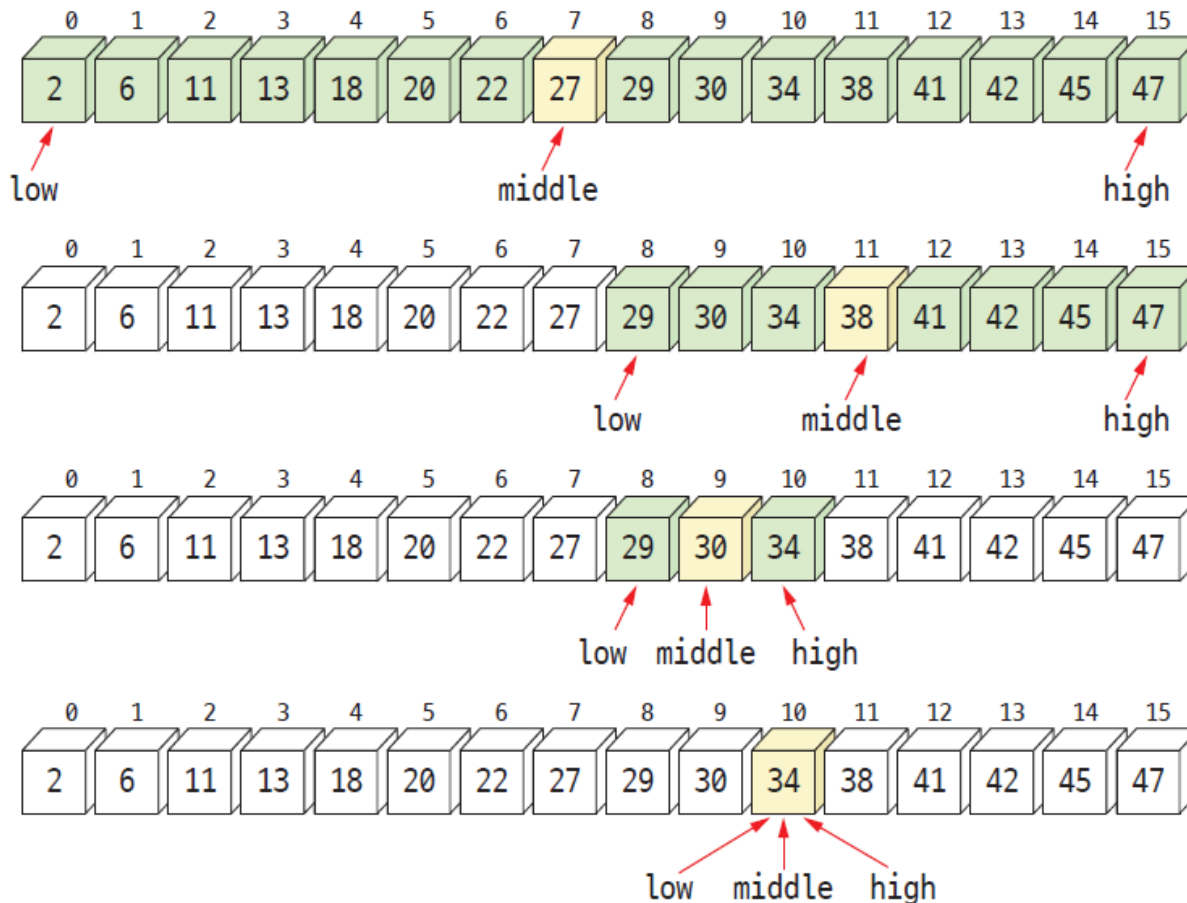
    printf ( " Search ended \n");
    return 0;
}
```

Using a for loop , the operation of comparing list[i] and key is repeated as many times as the size of the array. If list[i] and key are the same, the search is successful and the index of the array where the key value was found is printed .

Enter the value to search for :7  
Search Success Index = 6  
End of navigation

# Binary search

- Binary search : Repeated comparison with the element located at the center of a sorted array.



# Binary search

```
#include <stdio.h>
#define SIZE 16
int binary_search ( int list[], int n, int key);

int main( void )
{
    int key;
    int grade [SIZE] = { 2,6,11,13,18,20,22,27,29,30,34,38,41,42,45,47 };
    printf ( " Enter the value to search for : " );
    scanf ( "%d" , &key);
    printf ( " Search result = %d\n" , binary_search (grade, SIZE, key));

    return 0;
}
```

# Binary search

```
int binary_search ( int list[], int n, int key)
{
    int low, high, middle;
    low = 0;
    high = n-1;
    while ( low <= high ){ // If there are still numbers left
        printf ( "[%d %d]\n" , low, high); // Print the lower and upper limits .
        middle = (low + high)/2; // Calculate the middle position .
        if ( key == list[middle] ) // If matched, search succeeds
            return middle;
        else if ( key > list[middle] ) // If it is greater than the middle element
            low = middle + 1; // set
        else
            high = middle - 1; // set high to a new value
    }
    return -1;
}
```

# Execution results

Enter the value to search for :34

[0 15]

[8 15]

[8 10]

[10 10]

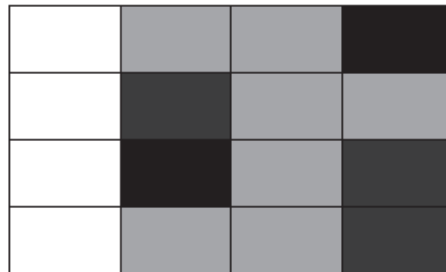
Search Results = 10



# 2-dimensional array

Used when storing data  
in table or matrix form.

- The data itself is often two-dimensional . For example, digital images or board games are fundamentally two-dimensional.

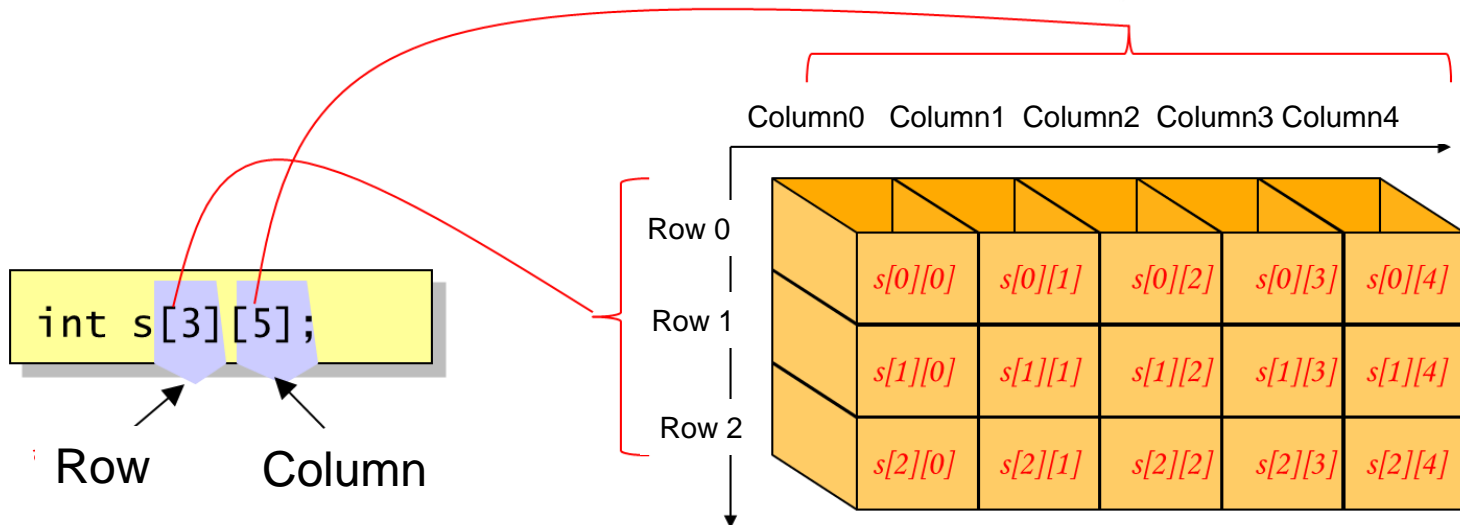


255	120	120	0
255	80	120	120
255	0	120	80
255	120	120	80

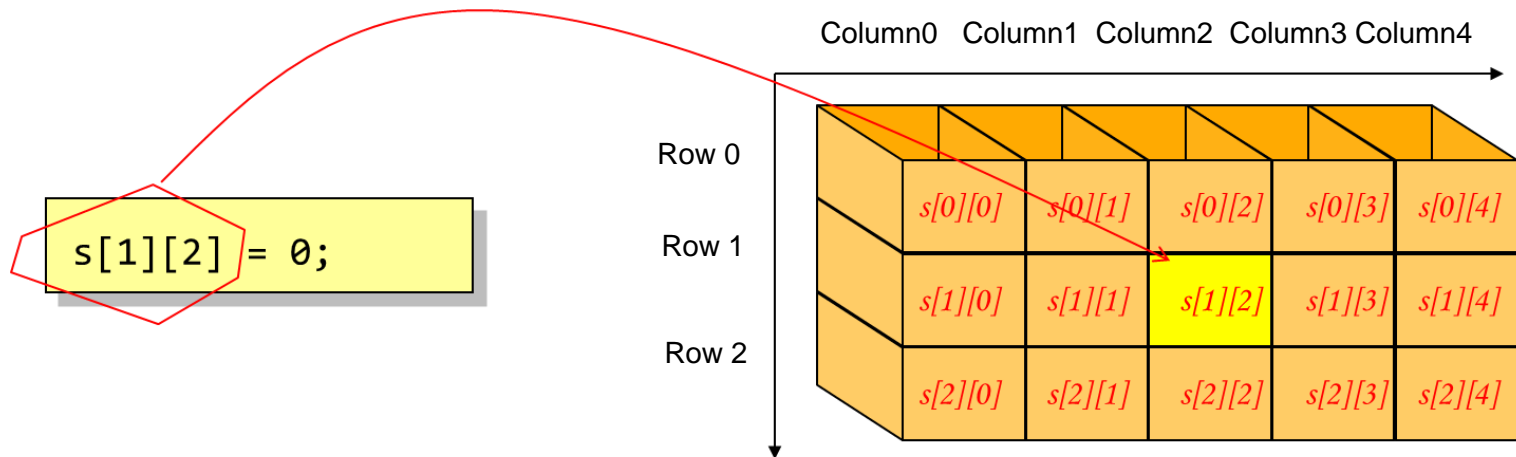
student number	Midterm Exam (30%)	Final exam (40%)	Final assignment (20%)	Quiz Score (10%)	Number of absences (point deduction)
1	87	98	80	76	3
2	99	89	90	90	0
3	65	68	50	49	0

# 2-dimensional array

```
int s[10]; // one-dimensional array  
int s[3][10]; // two-dimensional array  
int s[5][3][10]; // three-dimensional array
```



# Index in a 2-dimensional array



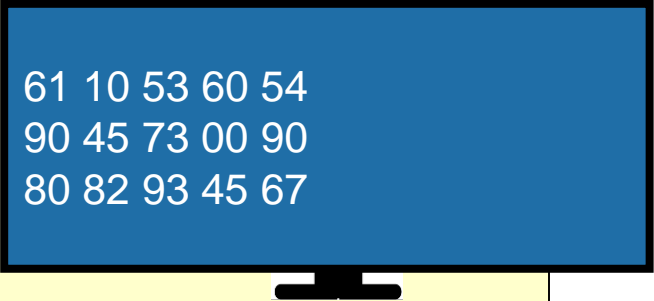
# Utilizing 2-dimensional arrays

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define ROWS 3
#define COLS 5

int main( void )
{
    int s[ ROWS ][ COLS ]; // Declare a two-dimensional array
    int i , j;              // 2 index variables
    srand (( unsigned )time(NULL)); // Initialize

    for (i = 0; i < ROWS ; i++)
        for (j = 0; j < COLS ; j++ )
            s[i][j] = rand() % 100;

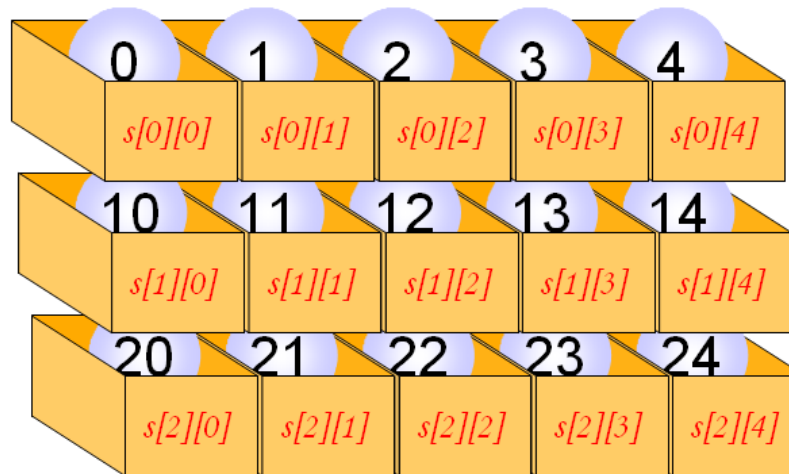
    for (i = 0; i < ROWS ; i++) {
        for (j = 0; j < COLS ; j++ )
            printf( " %02d " , s[i][j]);
        printf ( "\n" );
    }
    return 0;
}
```



```
61 10 53 60 54
90 45 73 00 90
80 82 93 45 67
```

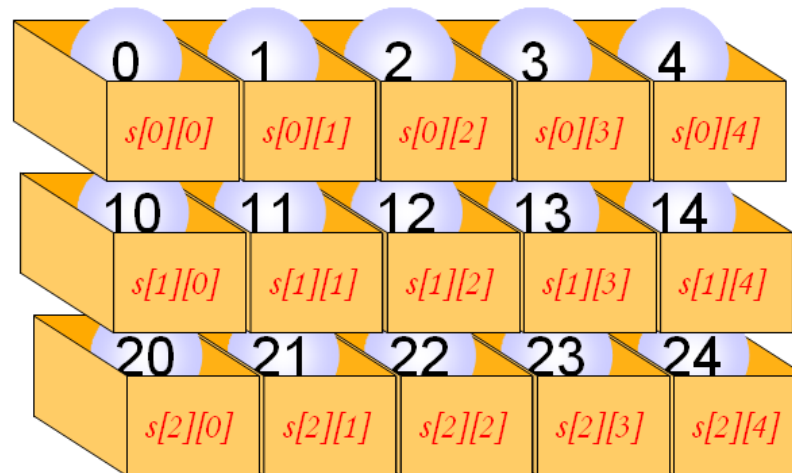
# Initializing a 2-dimensional array

```
int s[3][5] = {  
    { 0, 1, 2, 3, 4 },           // Initial values of elements in the first row  
    { 10, 11, 12, 13, 14 },    // Initial values of elements in the second row  
    { 20, 21, 22, 23, 24 }    // Initial values of elements in the third row  
};
```



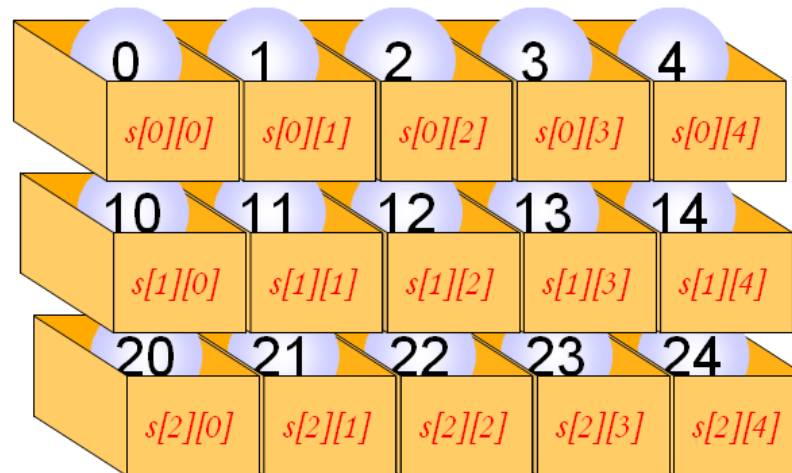
# Initializing a 2-dimensional array

```
int s[ ][5] = {  
    { 0,  1,  2,  3,  4 }, // Initial values of elements in the first row  
    { 10, 11, 12, 13, 14 }, // Initial values of elements in the second row  
    { 20, 21, 22, 23, 24 }, // Initial values of elements in the third row  
};
```



# Initializing a 2-dimensional array

```
int s[ ][5] = {  
0, 1, 2, 3, 4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24  
};
```



# Example



- The students' grade sheets in a two- dimensional array and calculate the final grade for each student .

Student ID	Midterm Exam (30%)	Final exam (40%)	Homework (20%)	Quiz Score (10%)	Absences (-1)
1	87	98	80	76	3
2	99	89	90	90	0
3	65	68	50	49	0

Final score =

Apply midterm exam 30%

Apply final exam 40%

Homework 20%

Quiz 10%

Deduct absences points



# Initializing a 2-dimensional array

```
#include <stdio.h>
#define ROWS 3
#define COLS 5
```

```
int main( void )
```

```
{
    int a[ ROWS ][ COLS ] = {
        { 87, 98, 80, 76, 3 },
        { 99, 89, 90, 90, 0 },
        { 65, 68, 50, 49, 0 }
    };

    int i ;
    for ( i = 0; i < ROWS ; i++ ) {
        double final_scores = a[ i ][0] * 0.3 + a[ i ][1] * 0.4 +
                               a[ i ][2] * 0.2 + a[ i ][3] * 0.1 - a[ i ][4];
        printf ( " Student #%i 's final grade =%10.2f \n" , i + 1, final_scores );
    }
    return 0;
}
```

Student #1 's final grade = 85.90  
Student #2 's final grade = 92.30  
Student #3 's final grade = 61.60

# Matrix

- The matrix is Used to solve many problems in natural science

$$A = \begin{bmatrix} 2 & 3 & 0 \\ 8 & 9 & 1 \\ 7 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 7 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Mathematics - ELEMENTARY MATRIX OPERATIONS

OPERATION: MULTIPLY EACH ELEMENT IN 2<sup>nd</sup> Row by 7:

$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$

1) FIND  $E = 2 \times 2 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 \\ 0 & 7 \end{bmatrix}$

$I \quad E$

2) PREMULT  $\begin{bmatrix} 1 & 0 \\ 0 & 7 \end{bmatrix} \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \Rightarrow \begin{bmatrix} 1(0)+0(3) & 1(1)+0(4) & 1(2)+0(5) \\ 0(0)+7(3) & 0(1)+7(4) & 0(2)+7(5) \end{bmatrix}$

# Representation of matrices using multidimensional arrays

[illegible]

# Representation of matrices using multidimensional arrays

```
int r, c;
```


```
// Add two matrices .
```

```
for (r = 0; r < ROWS; r++)  
    for (c = 0; c < COLS; c++)  
        C[r][c] = A[r][c] + B[r][c];
```

```
// Print the matrix .
```

```
for (r = 0; r < ROWS; r++)  
{  
    for (c = 0; c < COLS; c++)  
        printf ( "%d ", C[r][c]);  
    printf ( "\n" );  
}  
return 0;  
}
```

Using nested for loops, add each element of matrix A to each element of matrix B and assign it to matrix C.



3	3	0
9	9	1
8	0	5

# Passing a 2-dimensional array to a function

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

#define YEARS 3
#define PRODUCTS 5

int sum( int scores [ YEARS ][ PRODUCTS ] );

int main( void )
{
    int sales[ YEARS ][ PRODUCTS ] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
    int total_sale ;

    total_sale = sum(sales); // <--- &sales[0][0]
    printf ( " Total sales is %d .\n" , total_sale );

    return 0;
}
```

# Passing a 2-dimensional array to a function

```
int sum( int scores [ YEARS ][ PRODUCTS ])
{
    int y, p;
    int total = 0;

    for (y = 0; y < YEARS ; y++)
        for (p = 0; p < PRODUCTS ; p++)
            total += scores [y][p];

    return total;
}
```

*Total sales are 45 .*

# reference

## Note

In C, you can generally have n-dimensional arrays, such as two-dimensional arrays and three-dimensional arrays. In fact, there is no limit to the number of dimensions you can have in C.

```
int s[3][3][5]; // 3-dimensional array
```

However, you need to be careful because the amount of memory required increases drastically when it becomes multidimensional. It is usually better to avoid multidimensional arrays with three or more dimensions except in special cases. For example, a one-dimensional array that can store 100 integers only needs to be 400 bytes since one integer is 4 bytes, but a two-dimensional array of the size of 100 x 100 requires 40,000 bytes, and a three-dimensional array of the size of 100 x 100 x 100 requires 4,000,000 bytes. Therefore, you need to be careful not to increase the dimensions more than necessary.

# Lab: Image Processing

- A digital image can be thought of as a 2-dimensional array

```
int image[8][16] = {  
    { 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1 },  
    { 1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1 },  
    { 1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1 },  
    { 1,1,1,0,0,0,1,1,0,0,1,1,1,1,1,1 },  
    { 1,1,0,0,0,0,0,0,0,0,0,1,1,1,1,1 },  
    { 1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1 },  
    { 1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1 },  
    { 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1 } };
```





# Q & A

