

CYPRUS INTERNATIONAL UNIVERSITY  
ENGINEERING FACULTY

Lecture 3

Arrays

CMPE223 / ISYE223  
ALGORITHMS AND PROGRAMMING  
2023 – 2024 Spring

# Data Structures

- A single variable can be considered the simplest *data structure* (a fundamental one).
- A variable is a named storage location in the computer's memory where a value (integer number, floating point number, character, etc. ) can be stored and retrieved.

```
int x = 10;
```

`x` is a variable of type `int`, and stores the value `10`

- `x` can be considered a data structure because it **holds data** and **provides access** to that data.

However, *Data Structures* in programming refer to more complex data arrangements, such as *arrays*, *linked lists*, *trees*, *graphs*, etc., which allow for more sophisticated data manipulation and organisation.

# ARRAYS

- An Array stores multiple values using a **single name**.
- Subscripts or index values are used to distinguish between those different values.
- Data of the same kind (ages, temperatures, names, grades, etc.) can be stored in the memory as an array, making them easier to read and use.

# ARRAYS

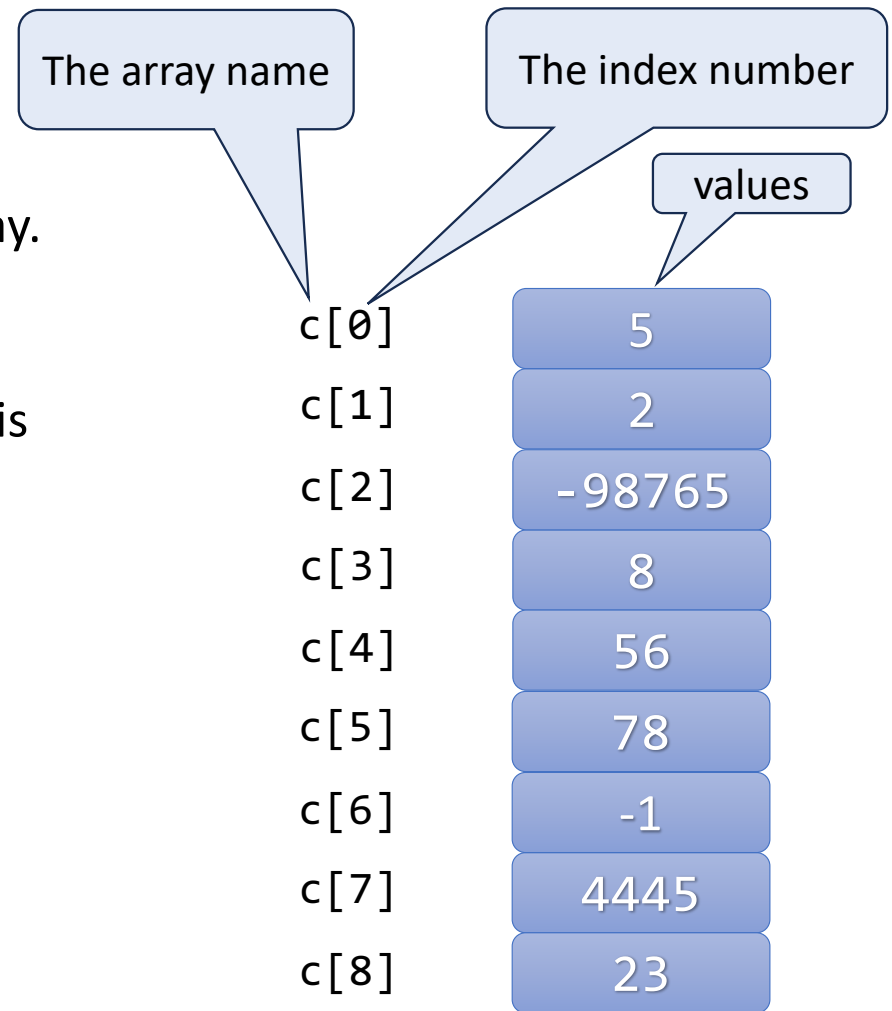
- An array is a series of variables of the same data type placed in consecutive memory locations as a block, with no other variable in the block.
- The programmer tells the computer how many memory locations to reserve for an array.
- Once the computer is told how many locations to reserve, that number cannot be changed while the program runs.
- The instructions for reserving memory for an array must be edited in the *source code*, and the program must be *re-compiled* and executed again.

# ARRAYS

- Each individual value of an array is called an *element* and is given a number (*index*) corresponding to its location in memory.
  - This is a reference number that gives an element's location information relative to the first member of the array (the actual memory address is stored with the array's name).
- An array element is written using two parts of information:
  1. The array name.
  2. The index number.

# One-Dimensional Array

- The simplest array is the *one-dimensional* array.
- Can be visualised as a column of memory locations.
- The index (number in the square `[]` brackets) is a reference number. It can be a *constant*, a *variable*, or an *expression*.



# One-Dimensional Array Declaration

All the elements in the array  
is of the same data type

Any valid variable name

The number of  
elements in the array.

```
data_type array_name[array_size];
```

The number of elements in an array **CAN NOT** change while the program is executing.

# Example:

An array named `numbers` that can store **five** values of type `int`

All the array elements are integers

array name

Array can store 5 integer values

```
int numbers[5];
```

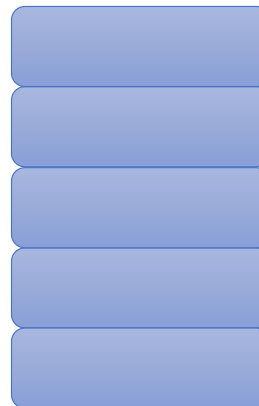
`numbers[0]`

`numbers[1]`

`numbers[2]`

`numbers[3]`

`numbers[4]`



Five sequential locations in memory. Each location can store an integer value.



# How to access array elements

Individual elements of an array are accessed using the array's **name** with an **index** enclosed in square brackets `[]`.

`array_name[index]`

- Each element in an array is numbered by its **index** (**offset/subscript**), which gives its relative position to the first element.
- The smallest index is called the **lower bound**; in C/C++, it is always **0**.
- The highest index is called the **upper bound**.
- Note that during **declaration**, the number in the square brackets `[]` is the **size** (range) of the array, but when an array element is **accessed**, the number denotes the element's **index**.

# Size of an Array

- The number of elements in the array is called its *range* (size of the array).

$$\begin{aligned}\text{range} &= \text{upper bound} - \text{lower bound} + 1 \\ \text{range} &= \text{upper bound} - 0 + 1\end{aligned}$$

lower bound = 0

$$\text{range} = \text{upper bound} + 1$$

The upper bound (highest index) is one less than the range(size) of the array.

# Example

```
int a[100]; // Specifies an array of 100 integers
```

The lower bound is 0, the range(size) of the array is 100, and the upper bound is 99 (range-1).

# Declaring and Initialising an Array (First Method)

`int num[5];`

size

**Declaration** of an array named `num` that contains `five` values of type `int`.

index

`num[0]=58;`

This is the **initialisation** of the **FIRST** array element with an integer value of `58`. The index is `0` (**lower bound**).

`num[1]=9;`

The initialisation of the **SECOND** (index is `1`) array element with an integer value of `9`.

`num[2]=8;`

The initialisation of the **THIRD** (index is `2`) array element with an integer value of `8`.

`num[3]=5;`

The initialisation of the **FOURTH** (index is `3`) array element with an integer value of `5`.

`num[4]=7;`

This is the **initialisation** of the **FIFTH** array element with an integer value of `7`. The index is `4` (**higher bound**).

# Declaring and Initialising an Array (Second Method)

```
float i[5];
```

Declaration of an array named `i` that contains `five` values of type `float`.

No size info is needed since it is specified during the declaration above.

```
i[] = { 58.2, 9.12, 8.77, 100.001, 7.19 };
```

The initialisation of all array elements with values of `58.2`, `9.12`, `8.77`, `100.001`, and `7.19` in sequence, starting with the first element (i.e. index `0`).

index `0`

index `1`

index `2`

index `3`

index `4`

## Second Method (cont'd)

Instead, we can declare and initialise on a single-line.

```
float i[5]={ 58.2, 9.12,8.77, 100.001, 7.19 };
```

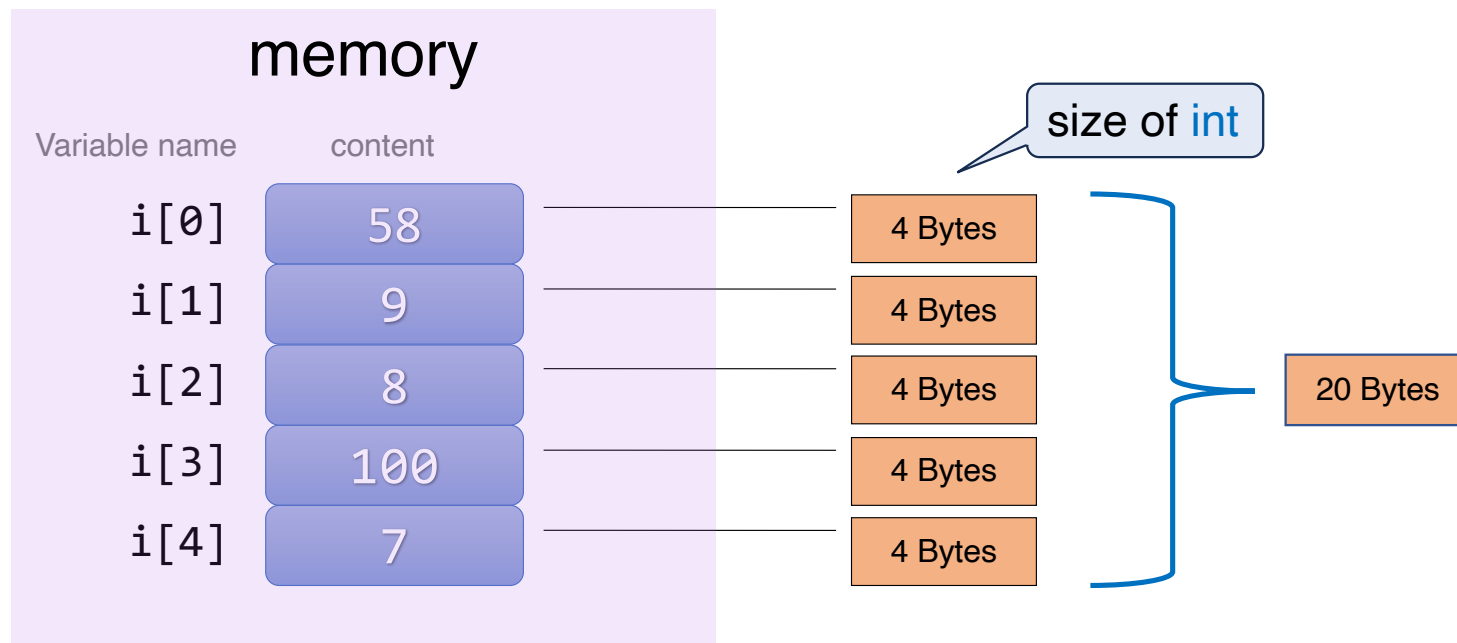
OR

```
float i[]={ 58.2, 9.12, 8.77, 100.001, 7.19 };
```

If the size info is not written by the programmer, then the compiler will count the actual number of elements and create an array to store them all.

# Visualising an Array in Memory

```
int i[]={ 58, 9, 8, 100, 7 };
```



# What happens ...

When a list of initialiser values is less than the number of array elements?

The remaining elements are initialised to **zero**.

index 0

index 1

```
double rate[3]={ 0.075, 0.81 };
```

Declaration and initialisation of an array named **rate** containing three values of type **double**, with the last array element initialised to **0.0** by default.

## memory

Variable name	content
rate[0]	0.075
rate[1]	0.81
rate[2]	0.0



# Example

```
int x[8]={3};
```

Declaration of an array named `x` containing `eight` values of type `int`, with the first array element initialised to value `3`.

## memory

Variable name	content
x[0]	3
x[1]	0
x[2]	0
x[3]	0
x[4]	0
x[5]	0
x[6]	0
x[7]	0

# Exercises

Declare an integer array named c with five elements.

```
int c[5];
```

Assign value 12 to the first element of the array c.

```
c[0] = 12;
```

Add the first 3 elements of array c and store the result in the 4th array element.

```
c[3] = c[0] + c[1] + c[2];
```

Divide the 4th element of array c by 7 and assign the result to the last element of array c.

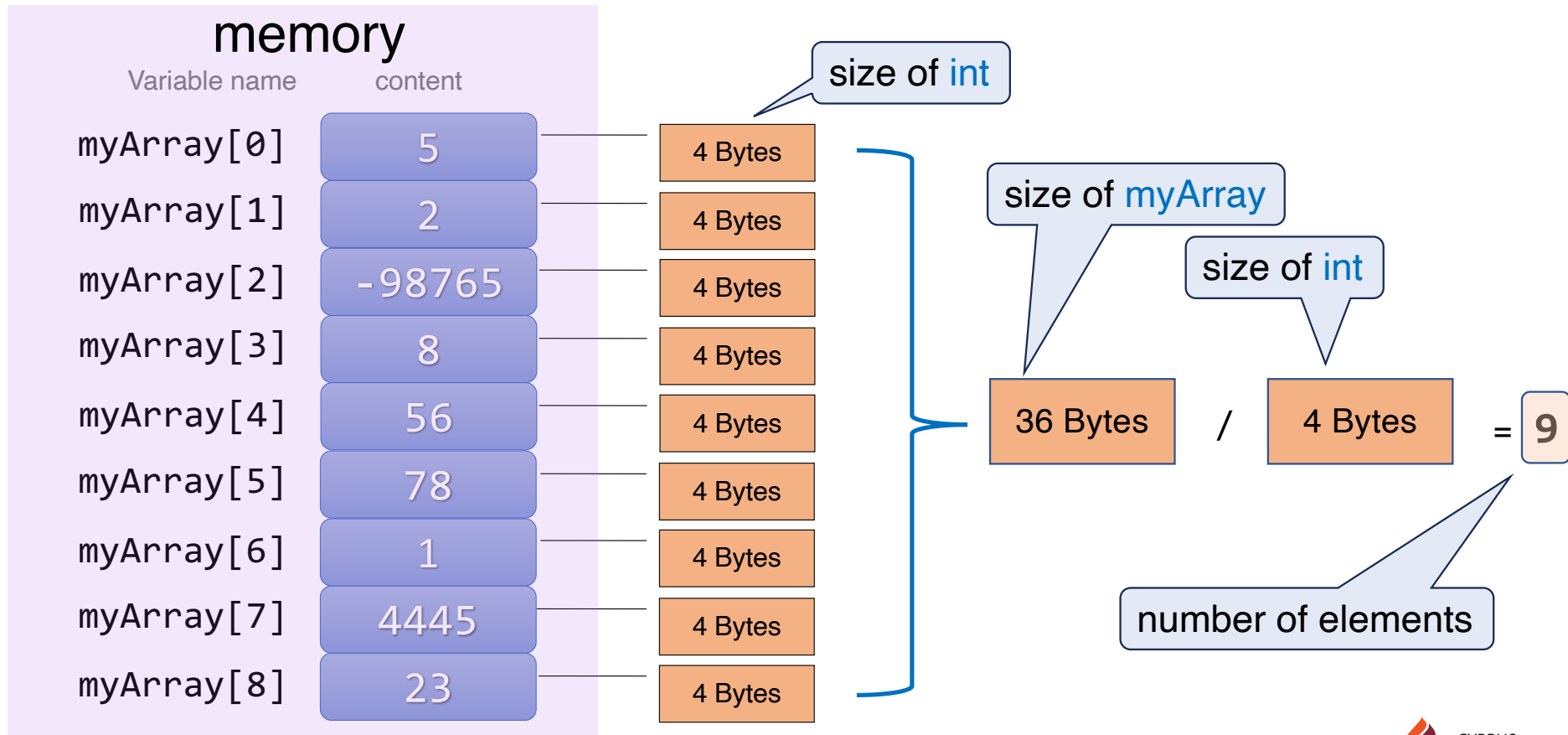
```
c[4] = c[3] / 7;
```

# Iterating Arrays

- Iterating through an array is performed using a loop.
- **Knowing or finding** the size of the array (number of elements) is necessary to determine the number of iterations that will be used for the loop.

# Finding the size of an array

```
int myArray[] = { 5, 2, -98765, 8, 56, 78, 1, 4445, 23 };
```



# sizeof Operator

- To find the size of an array (number of elements), use `sizeof` operator which gives the size of the operand in Bytes.

# Example

```
// Find the number of elements in array
#include <iostream>
using namespace std;

int main( void ){
    int myArray[] = { 5, 2, -98765, 8, 56, 78, 1, 4445, 23, 1209837, -9999, -834, 429, -
674, -388, -689, -654, 267, 301, 581, -387, -82, -402, -485, -649, 961, -911, -275, -41,
683, -846, 909, 284, 644, -618, 717, -862, -670, 594, -497, 729, -447, 497, -861, 389 };

    cout << "myArray is " << sizeof(myArray) << " Bytes" << endl;
    cout << "First element of myArray is " << sizeof(myArray[0]) << " Bytes" << endl;

    int array_size = sizeof(myArray)/sizeof(myArray[0]);
    cout << "myArray has " << array_size << " elements" << endl;

    for( int i=0; i<array_size; i++ )
        cout << "myArray[" << i << "] = " << myArray[i] << endl;

    return 0;
}
```

180 Bytes

4 Bytes

45 elements

# Two-Dimensional Arrays

- A two-dimensional array is a block of memory locations associated with a single variable name and designated by row and column numbers.
- The row number is always first, and the column number is second.
- The total number of elements in a two-dimensional array is calculated by multiplying the row size (first dimension) by the column size (second dimension).
- The row and column numbers can be *constants*, *variables*, or *expressions*, and they are of the integer data type.

# Two-Dimensional Array Declaration

All the elements in the array is of the same data type

Any valid variable name

The number of rows in the array.

The number of columns in the array.

```
data_type array_name[row_size][column_size];
```

Total Number of Elements =  $\text{row\_size} \times \text{column\_size}$

The number of elements in an array **CAN NOT** change while the program is executing.



# Two-Dimensional Array Declaration (cont'd)

An array named `numbers` that can store six values of type `int`

All the array elements are integers.

array name

The array can store 15 integer values.

```
int numbers[5][3];
```

column index

[0]

[1]

[2]

numbers[0]

[1]

[2]

[3]

[4]

row index

Fifteen locations in memory. Each location can store an integer value. Each element is referenced with a row and a column index.

# Iterating Two-Dimensional Arrays

- Iterating through a two-dimensional array is performed using nested loops.
- The outer loop is used for the rows, and the inner loop is used for the columns.

# Example

```
// Initialising the elements of a 2D array
#include <iostream>
using namespace std;
```

```
int main( void ){
    int d[3][3];
```

```
    d[0][0]=5;
    d[0][1]=9;
    d[0][2]=4;
```

First Row (index 0)

[0]

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

Second Row (index 1)

[1]

```
    d[2][0]=8;
    d[2][1]=6;
    d[2][2]=1;
```

Third Row (index 2)

[2]

First Column (index 0)	Second Column (index 1)	Third Column (index 2)
[0]	[1]	[2]
5	9	4
4	2	3
8	6	1

```
    for( int row=0; row<3; row++ ){
        for( int col=0; col<3; col++ )
            cout << d[row][col] << " ";
        cout << endl;
    }
    return 0;
```

```
}
```

# Re-writing The same Example

```
// Initialising the elements of a 2D array
#include <iostream>
using namespace std;
```

```
int main( void ){
    int b[3][3]={ {5,9,4}, {4,2,3}, {8,6,1} };
```

```
    for( int i=0; i<3; i++ ){
        for( int j=0; j<3; j++ )
            cout << b[i][j] << " ";
        cout << endl;
    }
```

```
    return 0;
```

```
}
```

	[0]	[1]	[2]
[0]	5	9	4
[1]	4	2	3
[2]	8	6	1

# What happens ...

When a list of initialiser values is less than the number of array elements?

The remaining elements are initialised to **zero**.

row index 0

row index 1

```
double rate[3][3]={ {0.075, 0.81}, {4.6} } ;
```

Declaration and initialisation of an array named **rate** containing three values of type **double**, with the last array element initialised to **0.0** by default.

## memory

Variable name	content		
	[0]	[1]	[2]
rate[0]	0.075	0.81	0.0
[1]	4.6	0.0	0.0
[2]	0.0	0.0	0.0

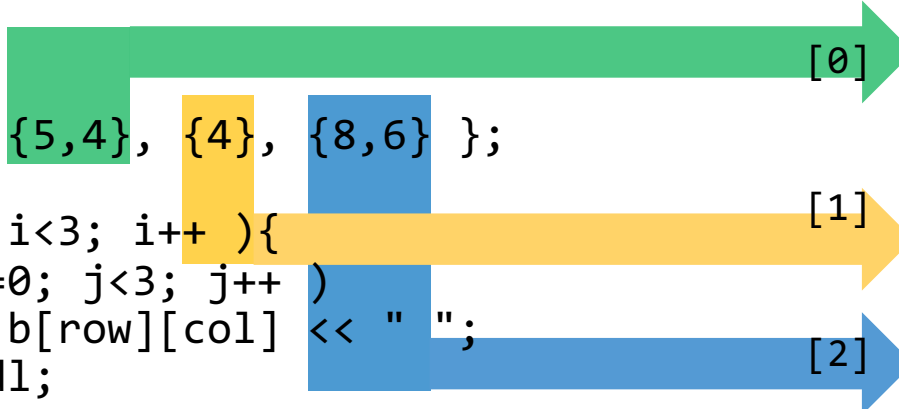
# Example

```
// Initialising the elements of a 2D array
#include <iostream>
using namespace std;
```

```
int main(void){
    int b[3][3]={ {5,4}, {4}, {8,6} };

    for( int i=0; i<3; i++ ){
        for( int j=0; j<3; j++ )
            cout << b[i][j] << " ";
        cout << endl;
    }

    return 0;
}
```



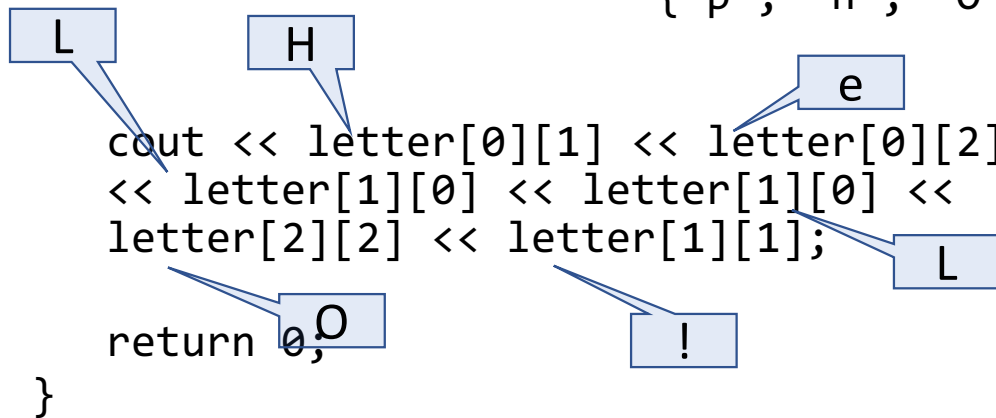
	[0]	[1]	[2]
[0]	5	4	0
[1]	4	0	0
[2]	8	6	0

# Example

```
#include <iostream>
using namespace std;
```

```
int main( void ){
    char letter[3][3] = { {'m', 'H', 'e'},
                          {'L', '!', 'E'},
                          {'p', 'n', 'O'} };
```

```
    cout << letter[0][1] << letter[0][2]
    << letter[1][0] << letter[1][0] <<
    letter[2][2] << letter[1][1];
    return 0;
}
```

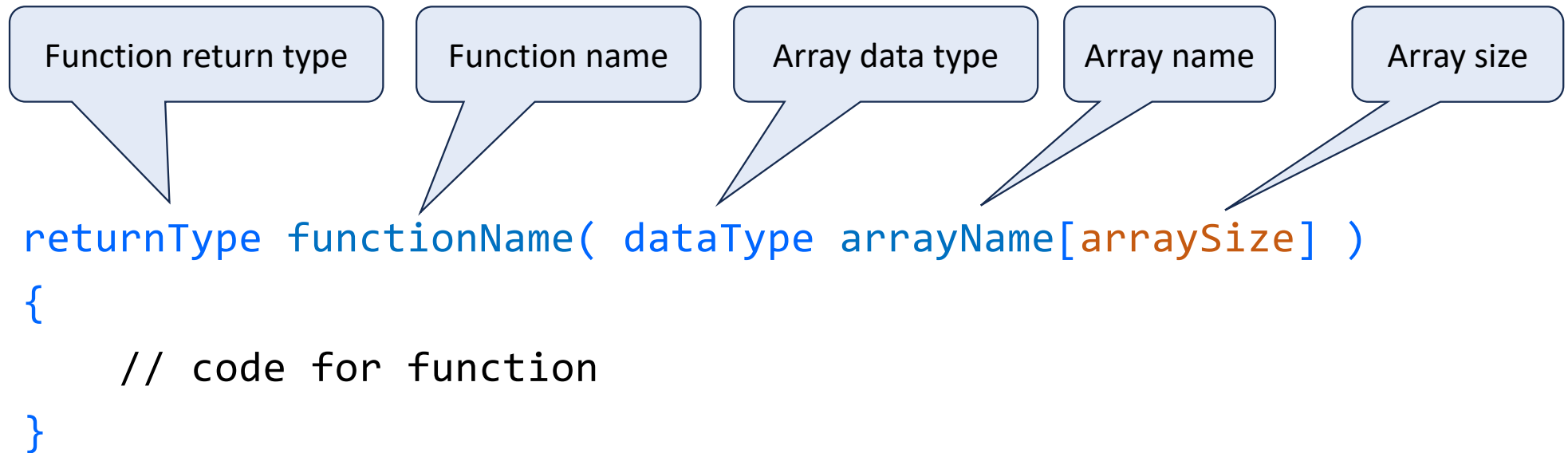


	[0]	[1]	[2]
[0]	m	H	e
[1]	L	!	E
[2]	p	n	O

OUTPUT

HeLLO!

# Passing an Array as a Parameter To a Function





# Example

```
// Passing an array to a function
#include <iostream>
using namespace std;

void display( int[], int );

int main( void ){
    int myArrray[5]= { 2, 10, 45, 5, 4 };

    display( myArrray, 5 );

    return 0;
}

void display( int a[], int size ){
    for( int i=0; i<size; i++ )
        cout << "a[" << i << "] = " << a[i] << endl;
}
```

Only the array's name is used when an array is passed as a (actual) parameter to a function. Square brackets **[ ]** are not used

Formal parameter definition must explicitly use square brackets **[ ]** to indicate that an array is expected from the calling module.

# Returning an Array From a Function

- C/C++ does **NOT** allow returning an array from a function.
- However, it is possible to return a *pointer* to an array by specifying its name without an index (no square brackets).

# OPERATIONS ON ARRAYS

Fill

Display

Copy

Delete

Insert

Sort

Search

Fill

and

Display

```
#include <iostream>
using namespace std;

int main( void ){
    int x[3];
    for(int i=0; i<3; i++)
        cin >> x[i];

    cout << endl << endl;

    for(int i=0; i<3; i++)
        cout << "x[" << i << "] = " << x[i] << endl;

    return 0;
}
```

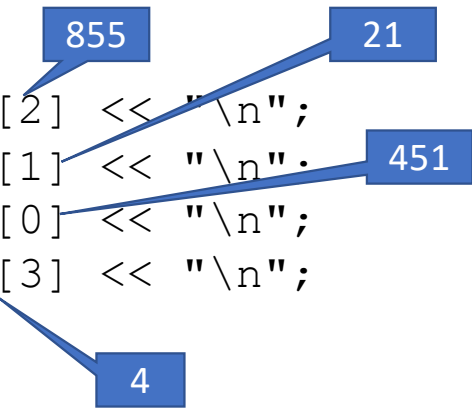
# Example

```
// Listing the elements of an array
#include <iostream>
using namespace std;
```

```
int main( void ){
    int a[4] = { 451, 21, 855, 4 };

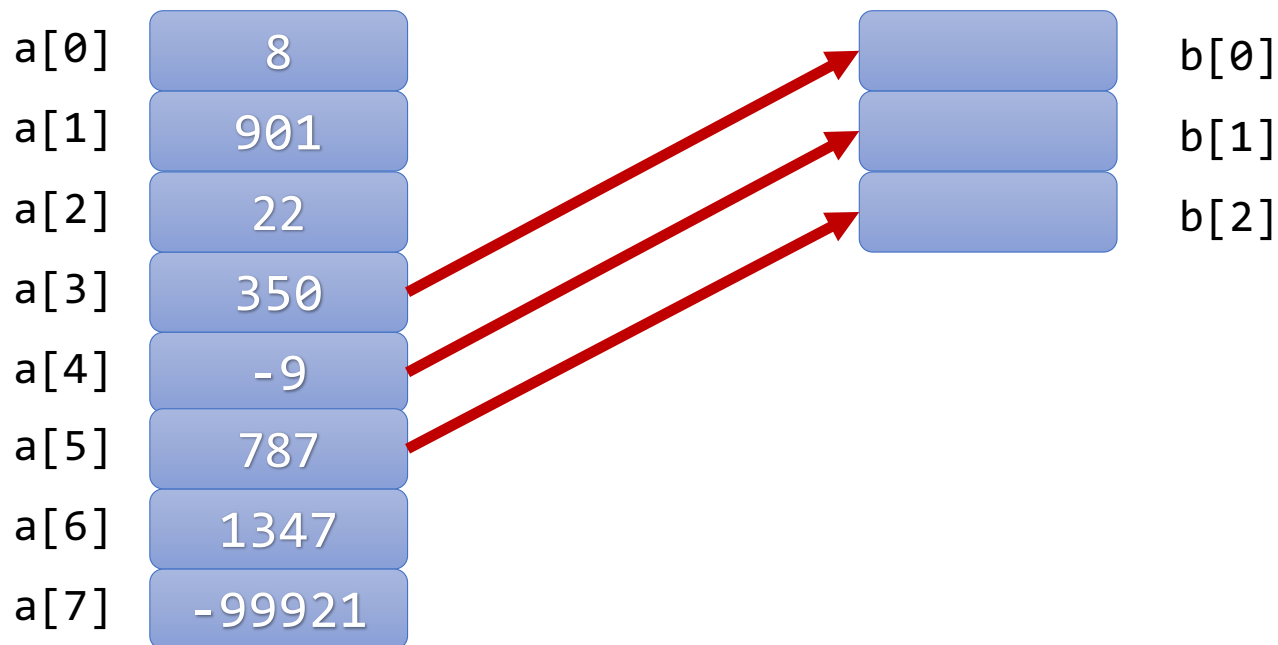
    cout << "Array element 2 : " << a[2] << "\n";
    cout << "Array element 1 : " << a[1] << "\n";
    cout << "Array element 0 : " << a[0] << "\n";
    cout << "Array element 3 : " << a[3] << "\n";

    return 0;
}
```



## Copy

- Assume that we have the following two arrays **a** and **b**. Lets copy 3 elements from **a** into **b**.



# COPYING

```
#include <iostream>
using namespace std;

int main( void ){
    int a[8] = { 8, 901, 22, 350, -9, 787, 1347, -99921 }, b[3];

    for(int i=0; i<8; i++)        // Display array a
        cout << "a[" << i << "] = " << a[i] << endl;

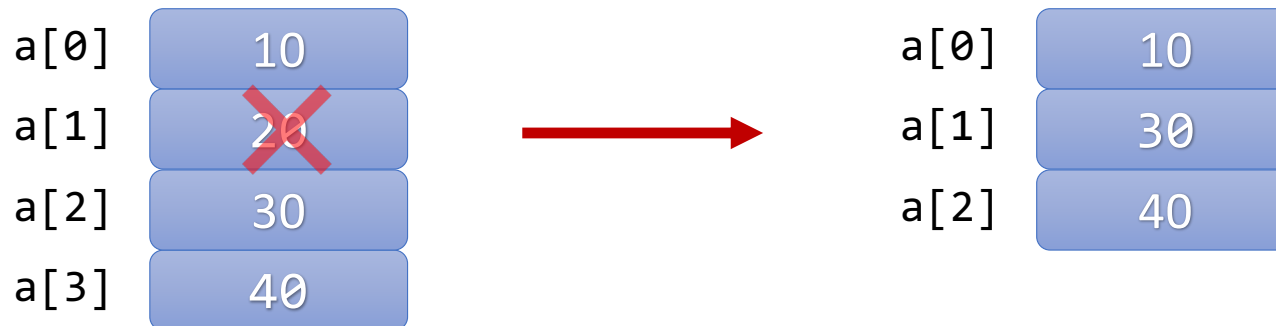
    for(int i=3; i<=5; i++)
        b[i-3] = a[i];           // Copy from array a to array b

    cout << endl;

    for(int i=0; i<3; i++)        // Display array b
        cout << "b[" << i << "] = " << b[i] << endl;
    return 0;
}
```

## Delete

- Assume that we have the following array `a`. Lets delete value 20 from `a`.





# DELETING

```
#include <iostream>
using namespace std;

int main( void ){
    int arr[] = {10, 20, 30, 40};
    int n = sizeof(arr) / sizeof(arr[0]); // Calculate the number of elements in the array

    // Print the original array
    cout << "Original Array: " << "\n";
    for (int i = 0; i < n; i++)
        cout << "arr[" << i << "] = " << arr[i] << endl;
    cout << "\n";

    // Find the index of the element to delete (if exists)
    int indexToDelete = -1;
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == 20)
        {
            indexToDelete = i;
            break;
        }
    }
}
```

# DELETING (cont'd)

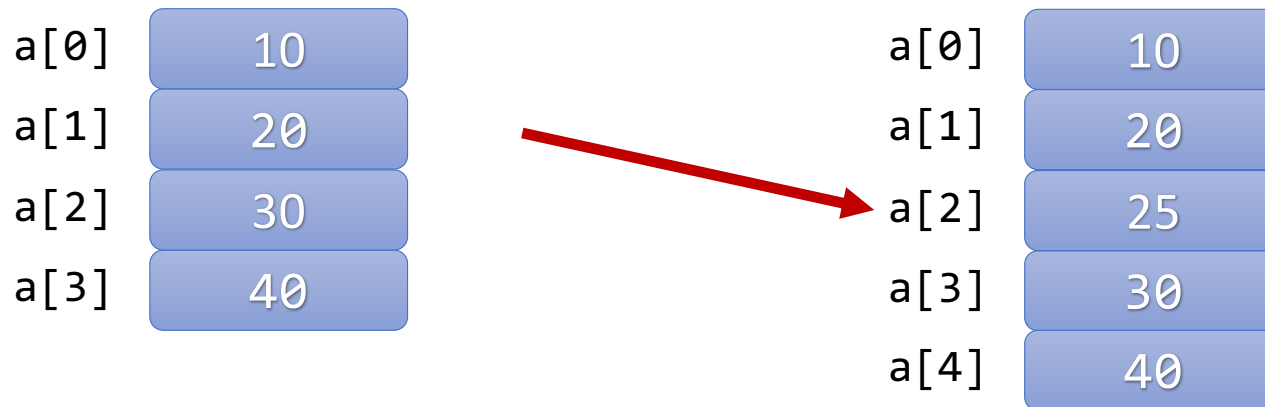
```
// If the element to delete is found, delete it by shifting elements
if (indexToDelete != -1)
{
    for (int i = indexToDelete; i < n - 1; i++)
    {
        arr[i] = arr[i + 1];
    }
    n--; // Decrease the size of the array
}

// Print the modified array after deletion
cout << "Array after deleting 20: \n";
for (int i = 0; i < n; i++)
{
    cout << "arr[" << i << "] = " << arr[i] << endl;
}
cout << "\n";

return 0;
}
```

## Insert

- Assume that we have the following sorted array **a**. Lets insert value 25 into **a**.



# INSERTING

```
#include <iostream>
using namespace std;

int main( void ){
    int arr[] = {10, 20, 30, 40}; // Define the original array with elements 10, 20, 30, and 40
    int originalSize = sizeof(arr) / sizeof(arr[0]);

    // Print the original array
    cout << "Original Array: " << "\n";
    for (int i = 0; i < originalSize; i++)
        cout << "arr[" << i << "] = " << arr[i] << endl;
    cout << "\n";

    // Define the new array to hold the inserted value
    int newArray[originalSize + 1];
    int newSize = 0;

    int insertValue = 25;
```

# INSERTING (cont'd)

```
// Copy elements from the original array to the new array until reaching the insertion point
int insertIndex = 0;
for (int i = 0; i < originalSize; i++)
{
    newArray[newSize++] = arr[i];
    if (arr[i] < insertValue && arr[i + 1] > insertValue)
    {
        insertIndex = newSize;
        newArray[newSize++] = insertValue; // Insert the value 25
    }
}

// Print the new array with the inserted value
cout << "Array after inserting " << insertValue << ": \n";
for (int i = 0; i < newSize; i++)
{
    cout << "arr[" << i << "] = " << newArray[i] << endl;
}
cout << "\n";

return 0;
}
```

# References

- Sprankle, M., & Hubbard, J. (2008). *Problem solving and programming concepts*. Prentice Hall Press.
- A Book on C, Fourth Edition, Al Kelley and Ira Pohl, Addison Wesley, 1999.
- C How to Program, Ninth Edition, Deitel & Deitel, Prentice Hall.