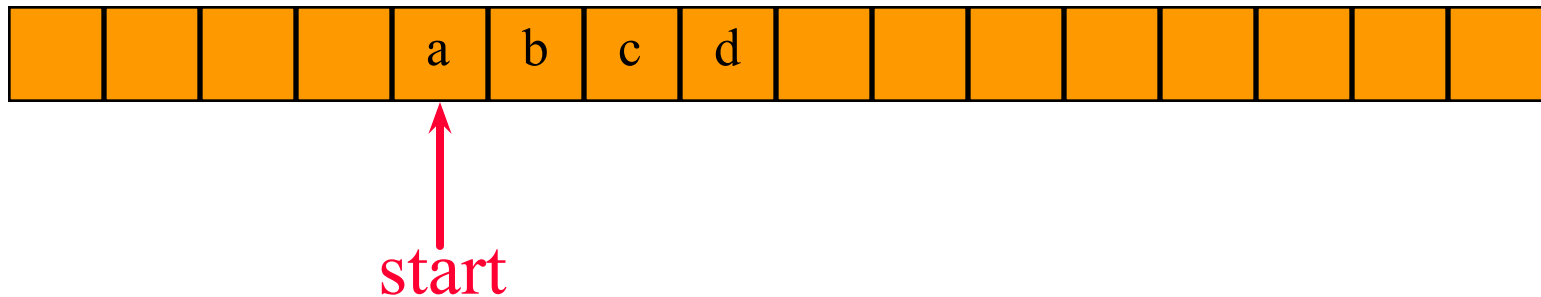


# Arrays



# 1D Array Representation In C++

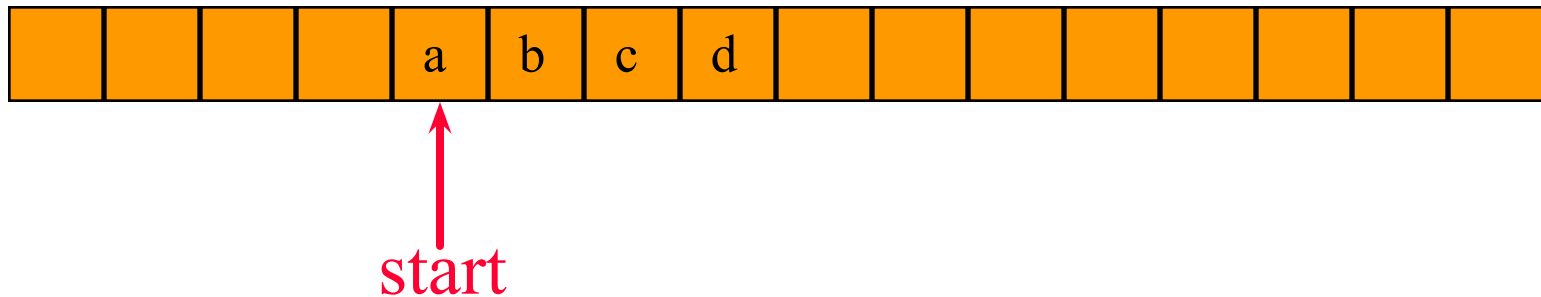
Memory



- 1-dimensional array  $x = [a, b, c, d]$
- map into contiguous memory locations
- $\text{location}(x[i]) = \text{start} + i$

# Space Overhead

Memory



space overhead = 4 bytes for **start**

(excludes space needed for the elements of **x**)

# 2D Arrays

The elements of a 2-dimensional array **a**  
declared as:

```
int [][]a = new int[3][4];
```

may be shown as a table

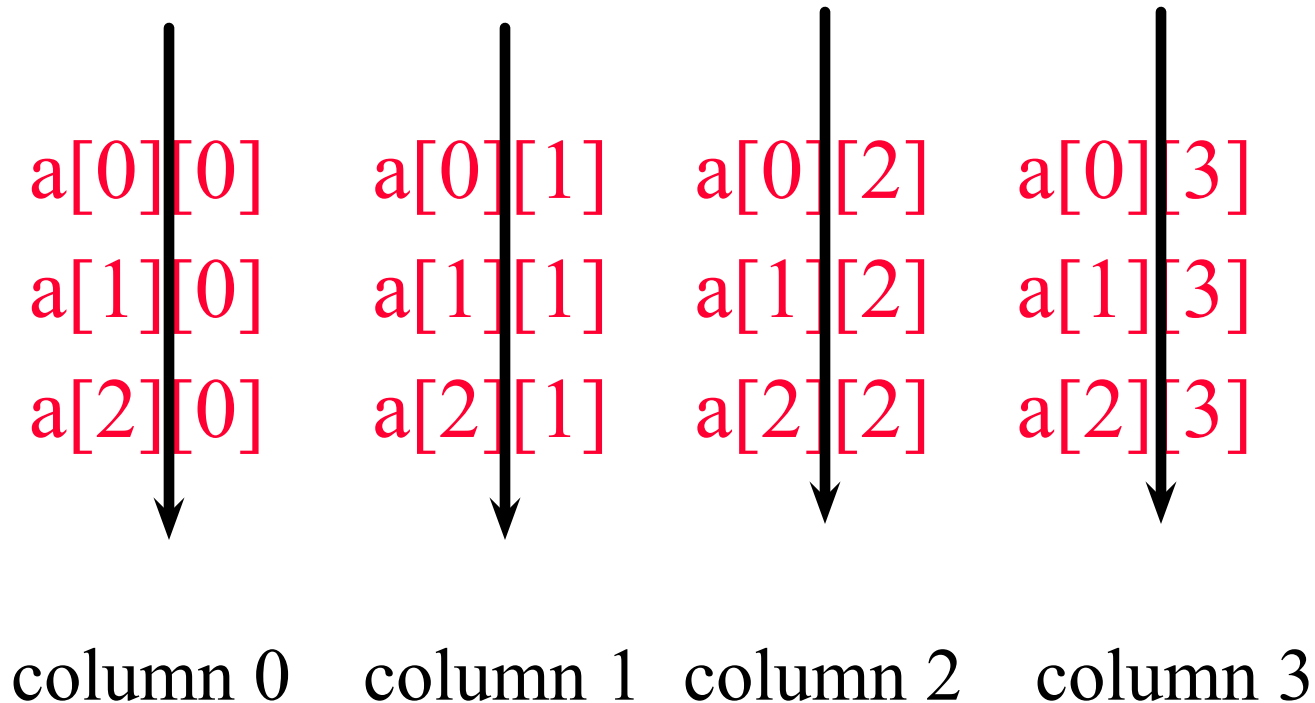
a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

# Rows Of A 2D Array

The diagram illustrates three rows of a 2D array. Each row is represented by a horizontal black line with an arrow pointing to the right. The elements of each row are labeled with red text as `a[row][col]`, where `row` is the row index and `col` is the column index. The rows are labeled on the right as 'row 0', 'row 1', and 'row 2'.

Row	Column 0	Column 1	Column 2	Column 3
row 0	<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>
row 1	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>
row 2	<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>

# Columns Of A 2D Array



# 2D Array Representation In C++

2-dimensional array **x**

a, b, c, d

e, f, g, h

i, j, k, l

view 2D array as a 1D array of rows

**x** = [row0, row1, row 2]

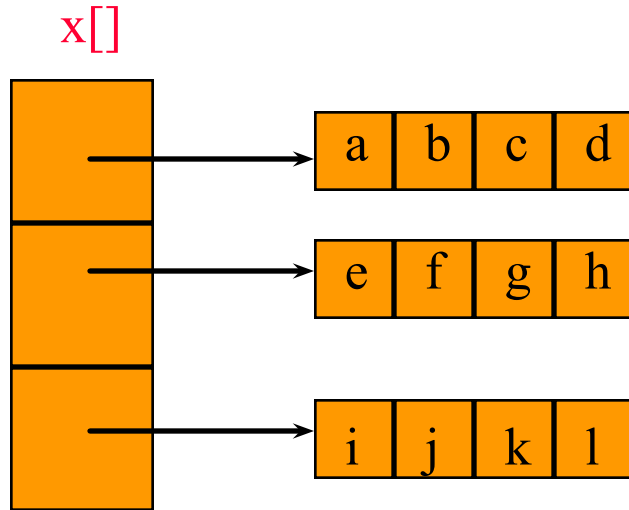
row 0 = [a,b, c, d]

row 1 = [e, f, g, h]

row 2 = [i, j, k, l]

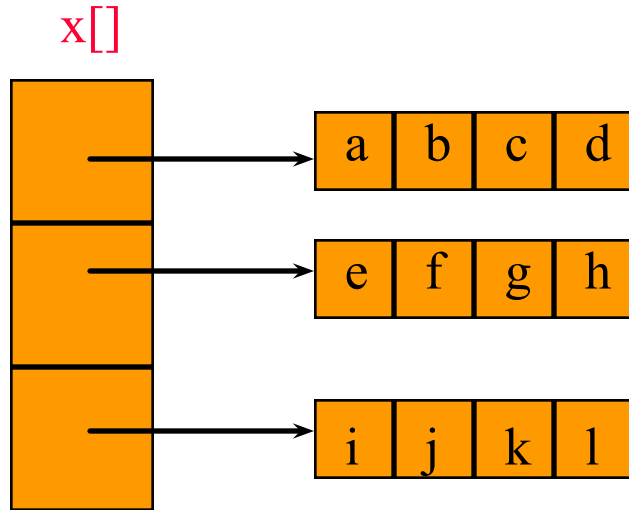
and store as **4** 1D arrays

# 2D Array Representation In C++



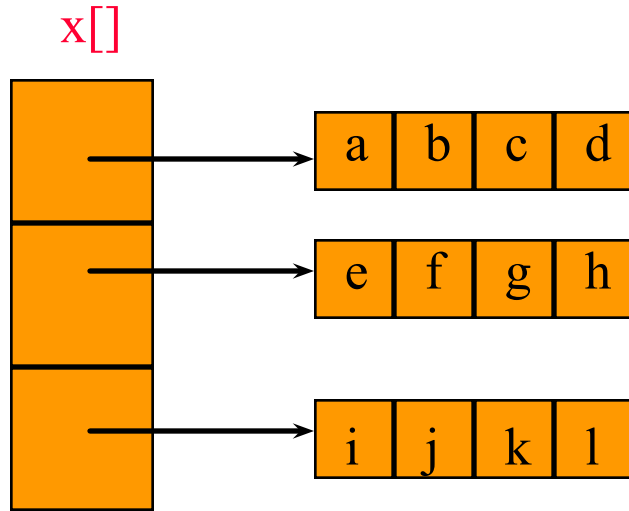


# Space Overhead



space overhead = overhead for 4 1D arrays  
=  $4 * 4$  bytes  
= 16 bytes  
= (number of rows + 1) x 4 bytes

# Array Representation In C++



- This representation is called the **array-of-arrays** representation.
- Requires contiguous memory of size 3, 4, 4, and 4 for the 4 1D arrays.
- 1 memory block of size **number of rows** and **number of rows** blocks of size **number of columns**

# Row-Major Mapping

- Example 3 x 4 array:

a b c d

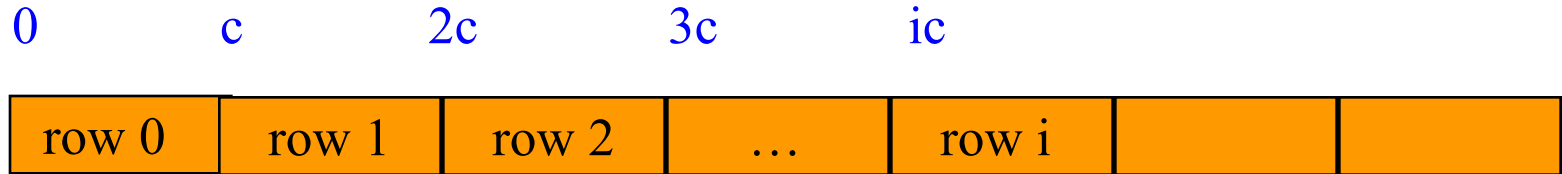
e f g h

i j k l

- Convert into 1D array  $y$  by collecting elements by rows.
- Within a row elements are collected from left to right.
- Rows are collected from top to bottom.
- We get  $y[] = \{a, b, c, d, e, f, g, h, i, j, k, l\}$

row 0	row 1	row 2	...	row i		
-------	-------	-------	-----	-------	--	--

# Locating Element $x[i][j]$



- assume  $x$  has  $r$  rows and  $c$  columns
- each row has  $c$  elements
- $i$  rows to the left of row  $i$
- so  $ic$  elements to the left of  $x[i][0]$
- so  $x[i][j]$  is mapped to position  
 $ic + j$  of the 1D array

# Space Overhead

row 0	row 1	row 2	...	row i		
-------	-------	-------	-----	-------	--	--

4 bytes for **start** of 1D array +  
4 bytes for **c** (number of columns)  
= 8 bytes

# Disadvantage

Need contiguous memory of size **rc**.

# Column-Major Mapping

a b c d

e f g h

i j k l

- Convert into 1D array  $y$  by collecting elements by columns.
- Within a column elements are collected from top to bottom.
- Columns are collected from left to right.
- We get  $y = \{a, e, i, b, f, j, c, g, k, d, h, l\}$

# Matrix

Table of values. Has rows and columns, but numbering begins at 1 rather than 0.

a b c d      row 1

e f g h      row 2

i j k l      row 3

- Use notation  $x(i,j)$  rather than  $x[i][j]$ .
- May use a 2D array to represent a matrix.



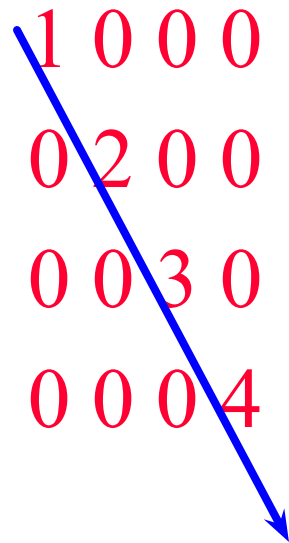
# Shortcomings Of Using A 2D Array For A Matrix

- Indexes are off by 1.
- C++ arrays do not support matrix operations such as **add**, **transpose**, **multiply**, and so on.
  - Suppose that **x** and **y** are 2D arrays. Can't do **x + y**, **x - y**, **x \* y**, etc. in Java.
- Develop a class **Matrix** for object-oriented support of all matrix operations.

# Diagonal Matrix

An  $n \times n$  matrix in which all nonzero terms are on the diagonal.

# Diagonal Matrix



A 4x4 diagonal matrix is shown with red numbers. The diagonal elements are 1, 2, 3, and 4. All other elements are 0. A blue arrow points from the top-left element (1) to the bottom-right element (4), indicating the diagonal.

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

- $x(i,j)$  is on diagonal iff  $i = j$
- number of diagonal elements in an  $n \times n$  matrix is  $n$
- non diagonal elements are zero
- store diagonal only vs  $n^2$  whole

# Lower Triangular Matrix

An  $n \times n$  matrix in which all nonzero terms are either on or below the diagonal.

1 0 0 0

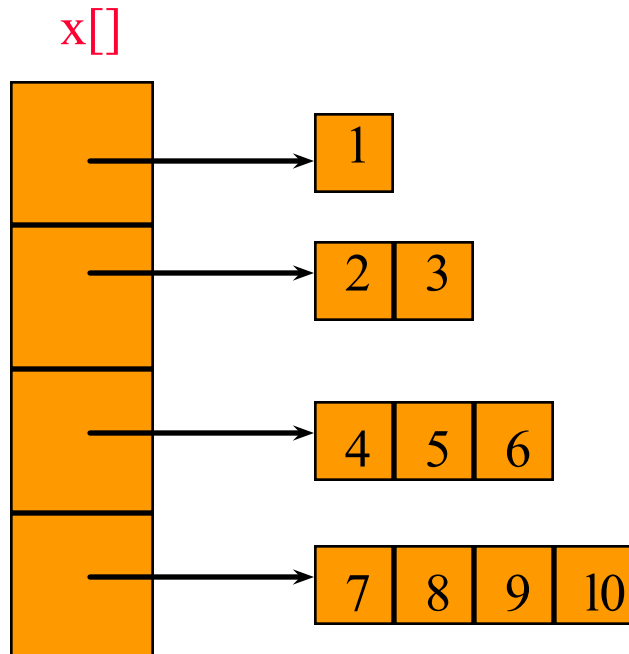
2 3 0 0

4 5 6 0

7 8 9 10

- $x(i,j)$  is part of lower triangle iff  $i \geq j$ .
- number of elements in lower triangle is  $1 + 2 + \dots + n = n(n+1)/2$ .
- store only the lower triangle

# Array Of Arrays Representation



Use an irregular 2-D array ... length of rows is not required to be the same.

# Creating And Using An Irregular Array

// declare a two-dimensional array variable

// and allocate the desired number of rows

```
int ** irregularArray = new int* [numberOfRows];
```

// now allocate space for the elements in each row

```
for (int i = 0; i < numberOfRows; i++)
```

```
    irregularArray[i] = new int [length[i]];
```

// use the array like any regular array

```
irregularArray[2][3] = 5;
```

```
irregularArray[4][6] = irregularArray[2][3] + 2;
```

```
irregularArray[1][1] += 3;
```

# Map Lower Triangular Array Into A 1D Array

Use row-major order, but omit terms that are not part of the lower triangle.

For the matrix

1 0 0 0

2 3 0 0

4 5 6 0

7 8 9 10

we get

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

# Index Of Element [i][j]

0	1	3	6			
r 1	r2	r3	...	row i		

- Order is: row 1, row 2, row 3, ...
- Row i is preceded by rows 1, 2, ..., i-1
- Size of row i is i.
- Number of elements that precede row i is  
 $1 + 2 + 3 + \dots + i-1 = i(i-1)/2$
- So element (i,j) is at position  $i(i-1)/2 + j - 1$  of the 1D array.



# Data Structure and Algorithm (UCS2004)

## B. Tech -2<sup>nd</sup> Semester

### Assignment -1

1. Define data structure and also describe the difference between primitive and non primitive data structure.
2. Discuss the importances of abstract data type (ADT) in programming?
3. Write a C program for insertion of an element in an Array?
4. Consider the linear arrays **AAA[5:50]**, **BBB[-5:10]** and **CCC[1:8]**
  - a) Find the number of elements in each array?
  - b) Suppose base (**AAA**) = **300** and **w=4** words per memory cell for AAA. Find the address of **AAA[15]**, **AAA[35]** and **AAA[55]**.
5. What is sparse matrix? Explain. Write the applications of sparse matrix?
6. Demonstrate and convert the infix expression **((A+B)\*(C-D))/E-(F\*(G+H)/I+J\*K)** to postfix expression using stack.
7. Write a C program to implement insertion of element at beginning, end and specific node in linked list.
8. Write algorithm to convert a postfix expression into an infix expression. Consider the following arithmetic expression in postfix notation:  
$$752+* 4 1 5- /-$$
  - (i) Find the value of the expression.
  - (ii) Find the equivalent prefix form of the above expression.
9. Provide a C program that implements these operations (**Push, Pop, Peek, and IsEmpty.**) using an **array-based stack**.

10. What is a **Priority Queue**? How does it differ from a regular queue? Write a C program to implement a priority queue using an array.
11. Discuss the implementation of single-linked list. Write C function to implement following operations on singly-linked list.
  - a. To count number of nodes
  - b. To reverse the direction of links.
  - c. To delete alternate nodes that is first, third, fifth and so on.
12. Write an algorithm to convert a valid arithmetic infix expression into its equivalent postfix expression.



# Linked Lists



- list elements are stored, in memory, in an arbitrary order
- explicit information (**called a link**) is used to go from one element to the next

# Memory Layout

Layout of  $L = (a,b,c,d,e)$  using an array representation.

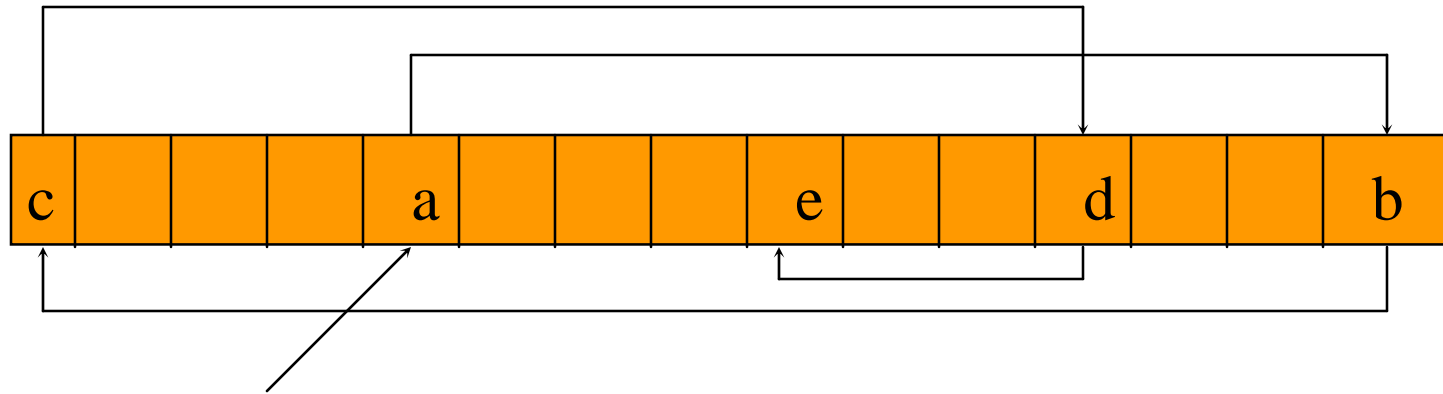


A linked representation uses an arbitrary layout.





# Linked Representation

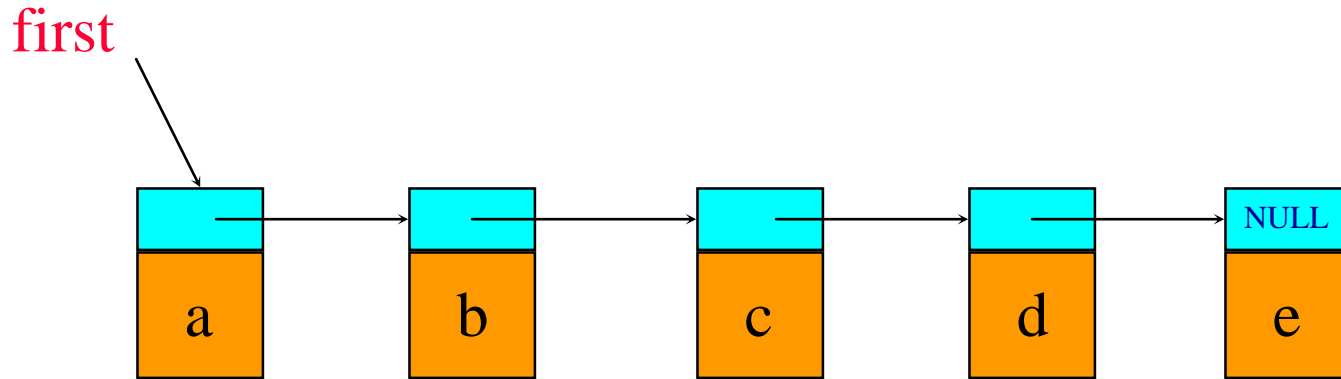


first

pointer (or link) in **e** is **NULL**

use a variable **first** to get to the first  
element **a**

# Normal Way To Draw A Linked List

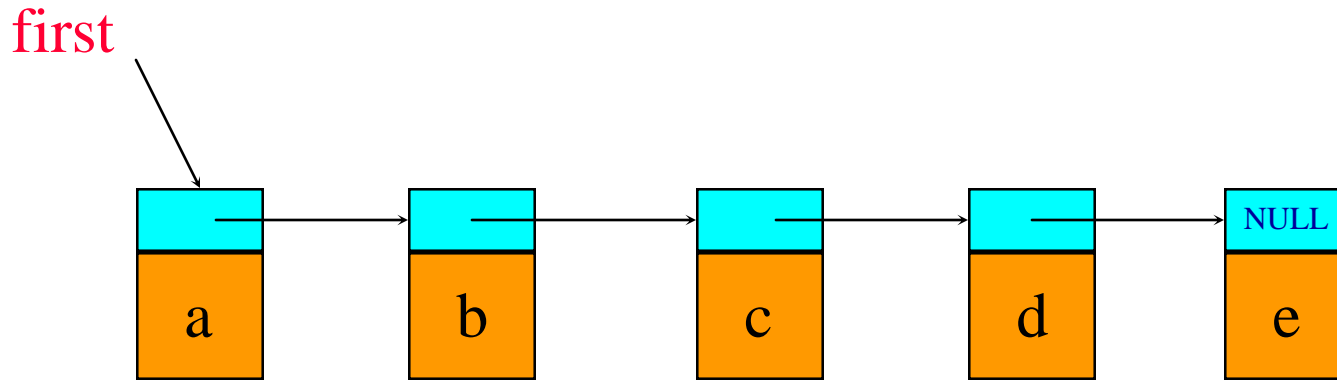


link or pointer field of node



data field of node

# Chain

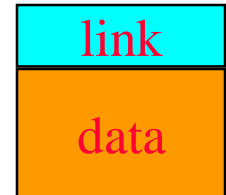


- A chain is a linked list in which each node represents one element.
- There is a link or pointer from one element to the next.
- The last node has a **NULL** (or **0**) pointer.

# Node Representation

```
template <class T>
class ChainNode
{
    private:
        T data;
        ChainNode<T> *link;

        // constructors come here
};
```

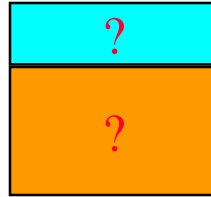




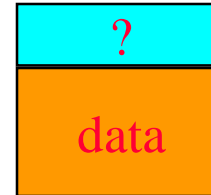
# Constructors Of ChainNode



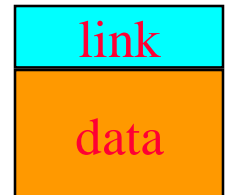
```
ChainNode() {}
```



```
ChainNode(const T& data)  
{ this->data = data; }
```

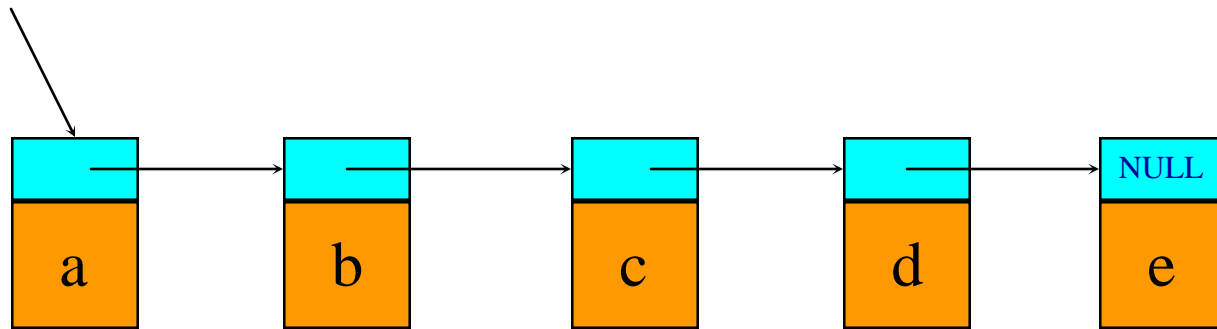


```
ChainNode(const T& data, chainNode<T>* link)  
{ this->data = data;  
  this->link = link; }
```



# Get(0)

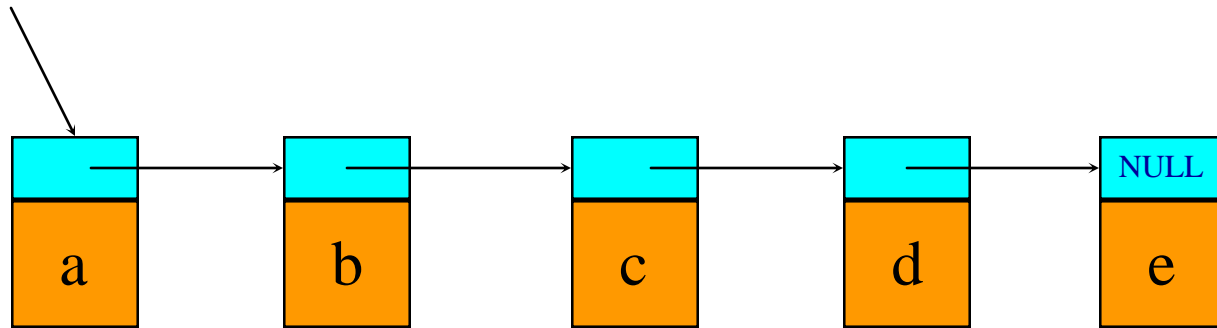
first



```
desiredNode = first; // gets you to first node  
return desiredNode->data;
```

# Get(1)

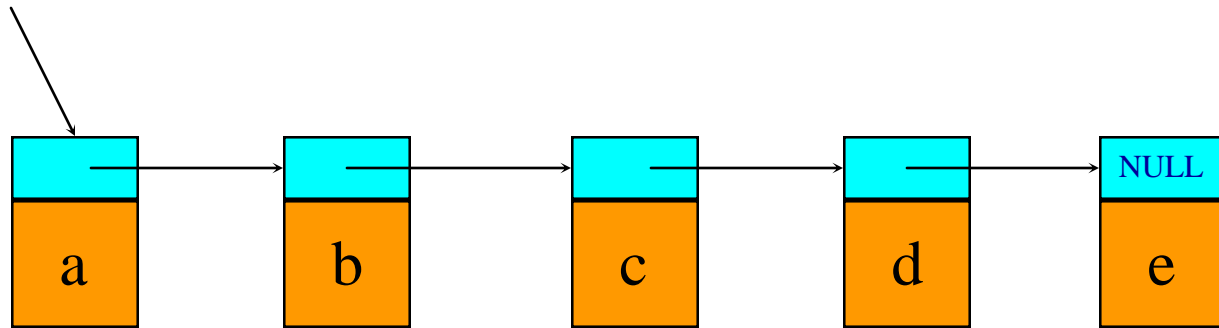
first



```
desiredNode = first->link; // gets you to second node  
return desiredNode->data;
```

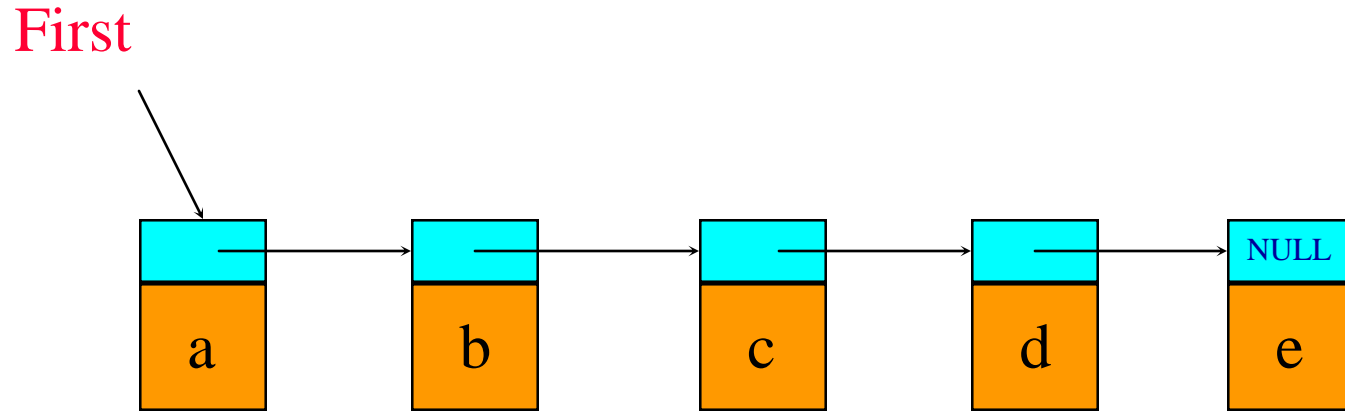
## Get(2)

first



```
desiredNode = first->link->link; // gets you to third node  
return desiredNode->data;
```

# Get(5)

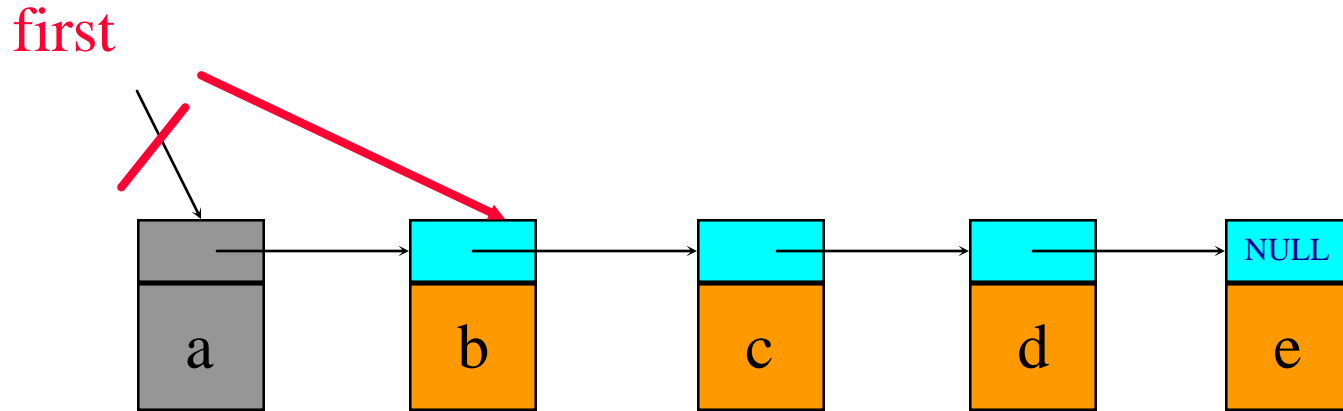


```
desiredNode = first-link-link-link-link-link;
```

```
// desiredNode = NULL
```

```
return desiredNode-data; // NULL.element
```

# Delete An Element



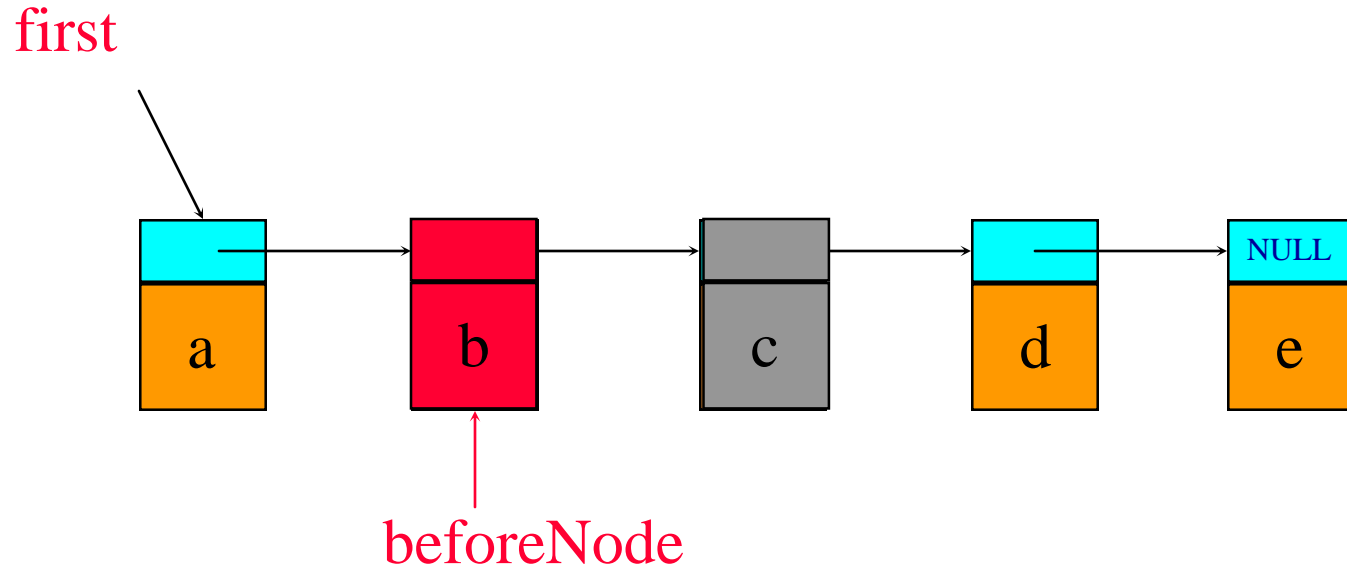
Delete(0)

`deleteNode = first;`

`first = first-•link;`

`delete deleteNode;`

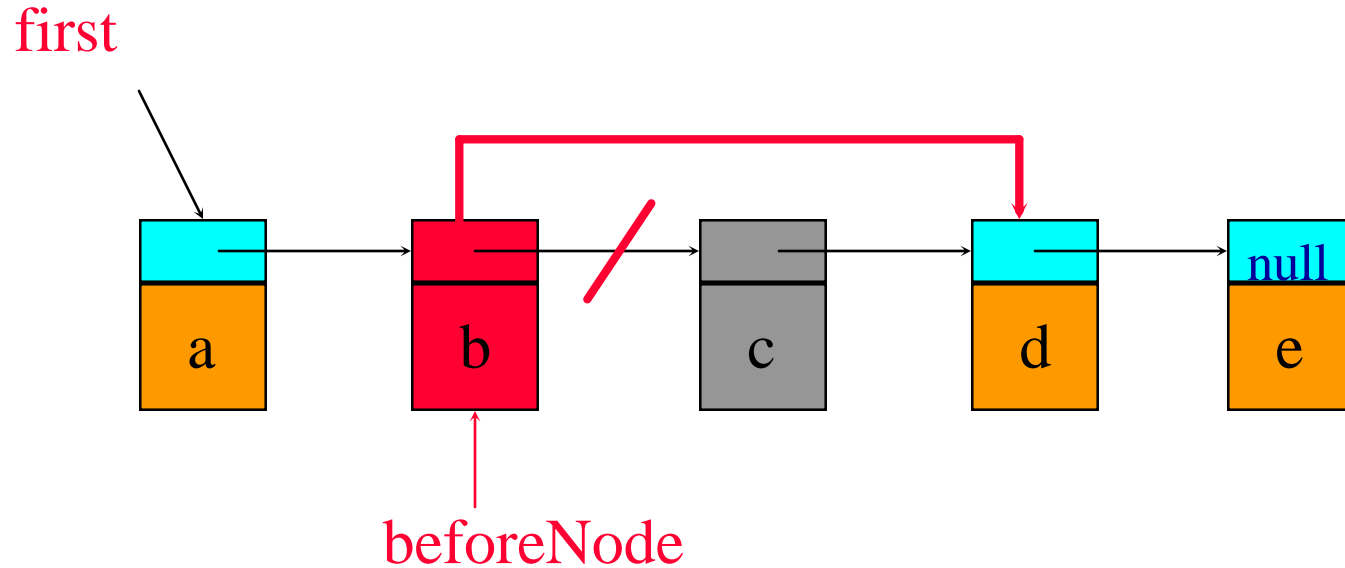
## Delete(2)



first get to node just before node to be removed

$\text{beforeNode} = \text{first} \rightarrow \text{link};$

## Delete(2)

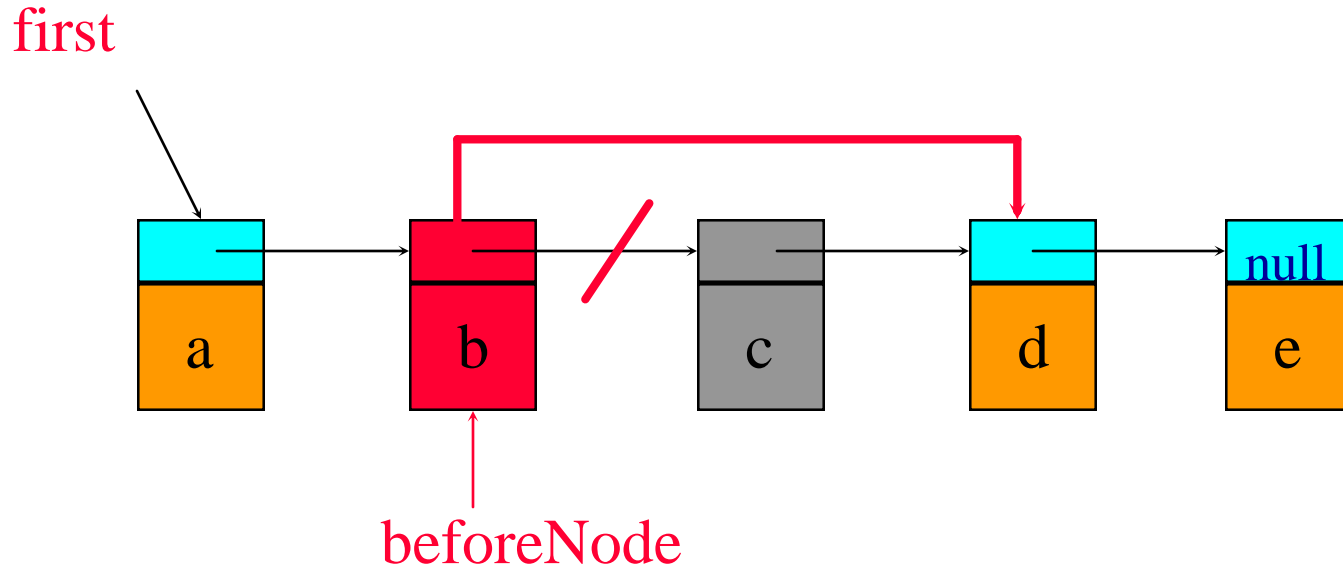


save pointer to node that will be deleted

`deleteNode = beforeNode->link;`



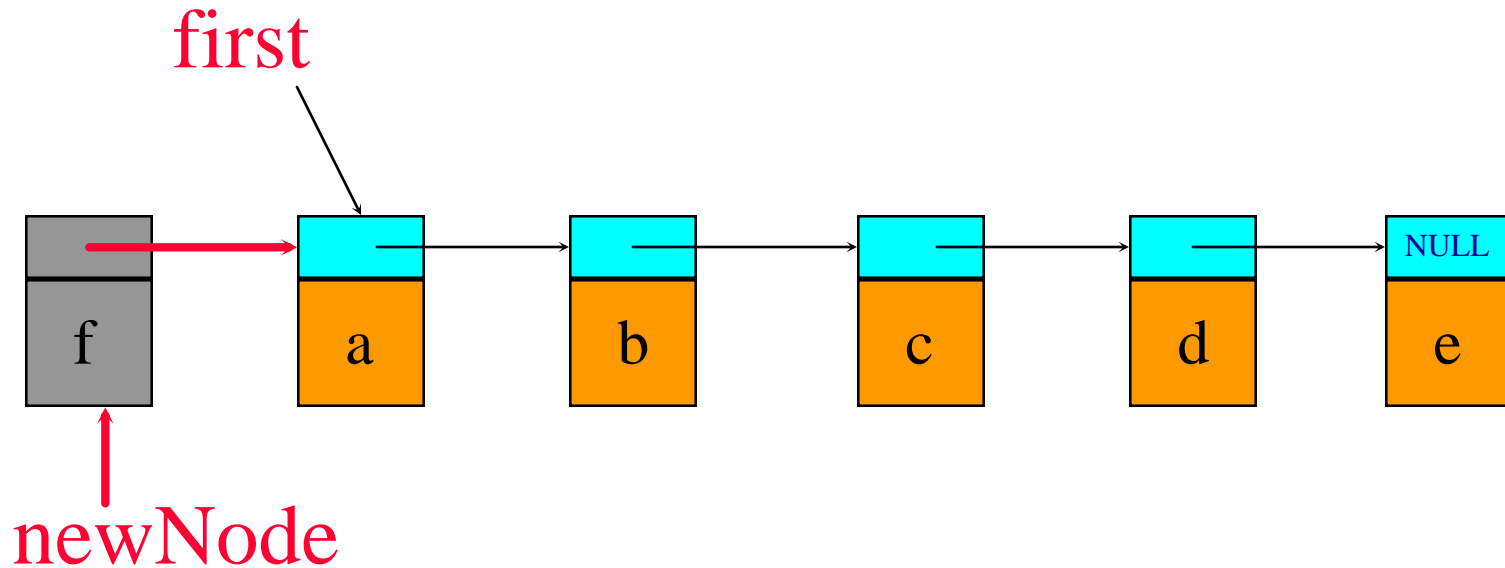
## Delete(2)



now change pointer in **beforeNode**

**beforeNode-•link = beforeNode-•link-•link;**  
**delete deleteNode;**

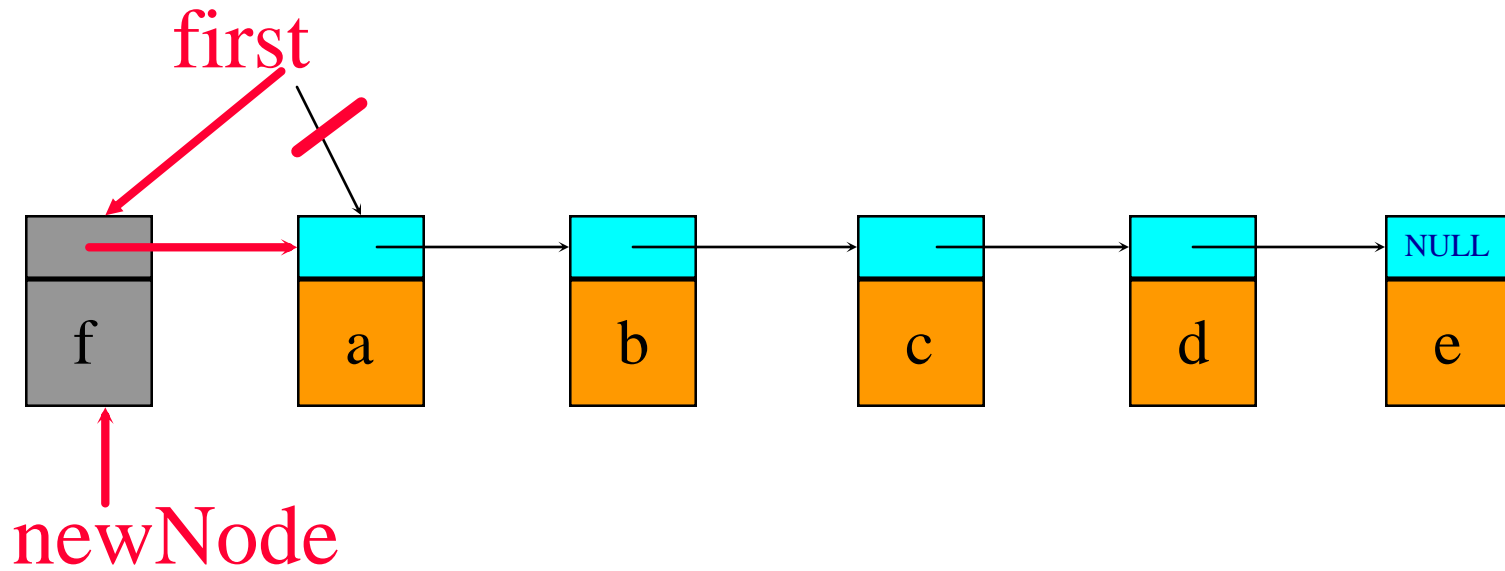
# Insert(0, 'f')



Step 1: get a node, set its data and link fields

```
newNode = new ChainNode<char>(theElement,  
                                first);
```

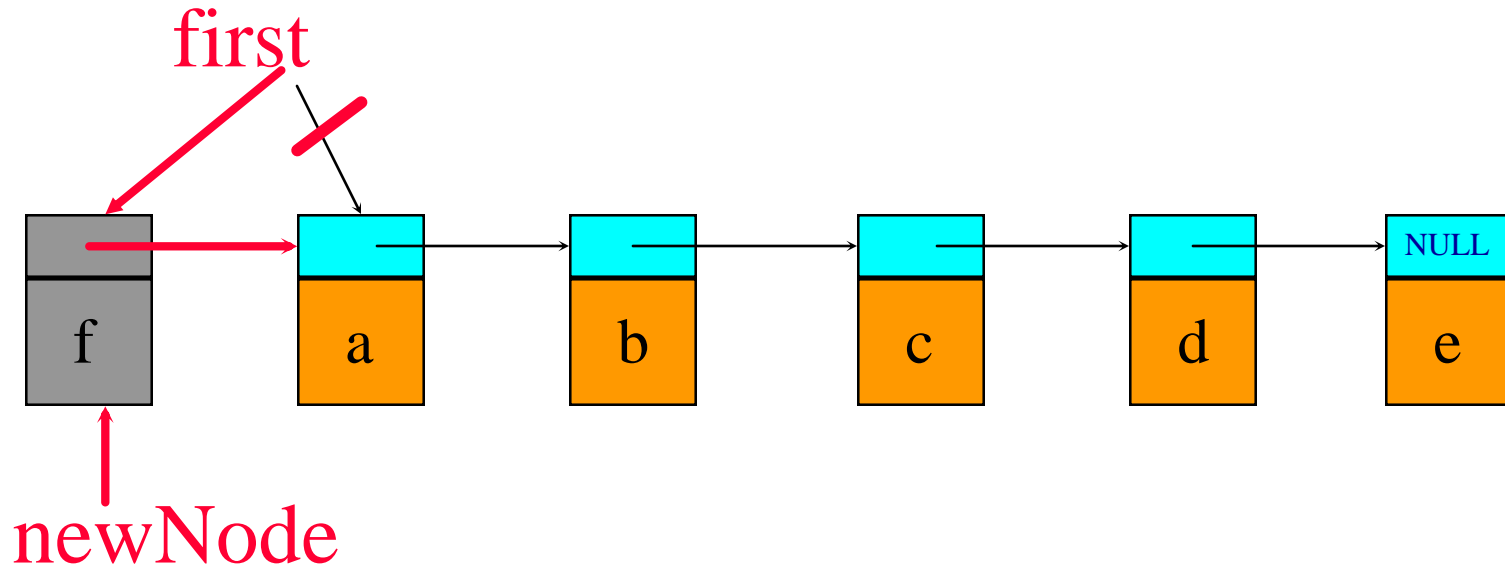
# Insert(0, 'f')



Step 2: update **first**

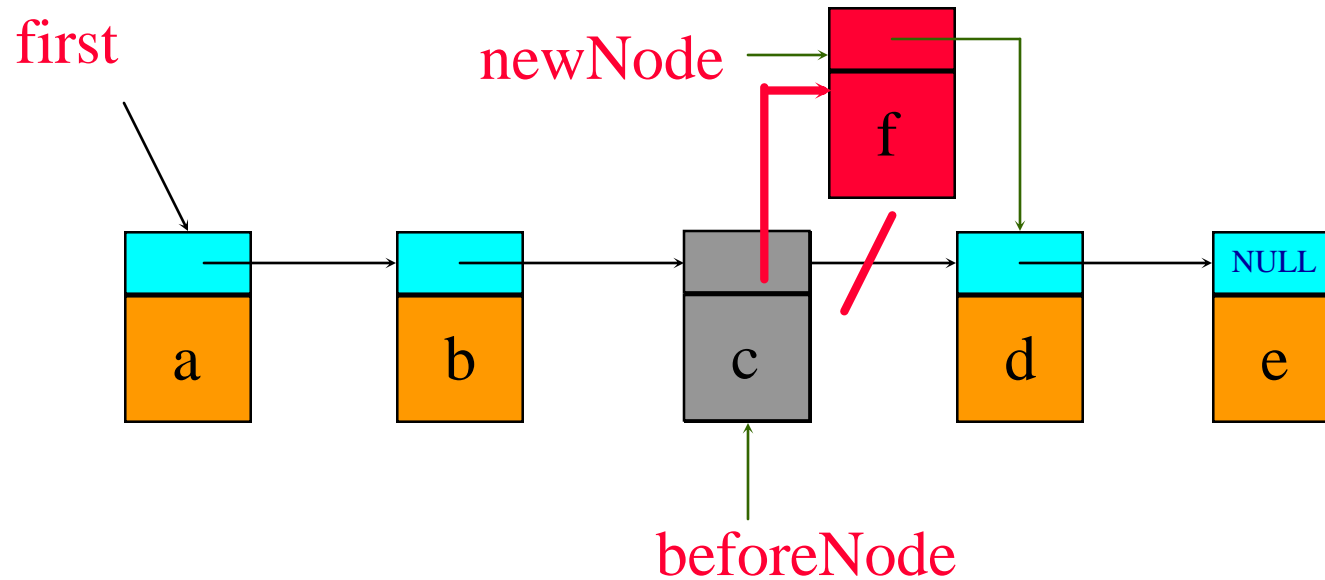
**first = newNode;**

# One-Step Insert(0, 'f')



```
first = new chainNode<char>('f', first);
```

# Insert(3, 'f')

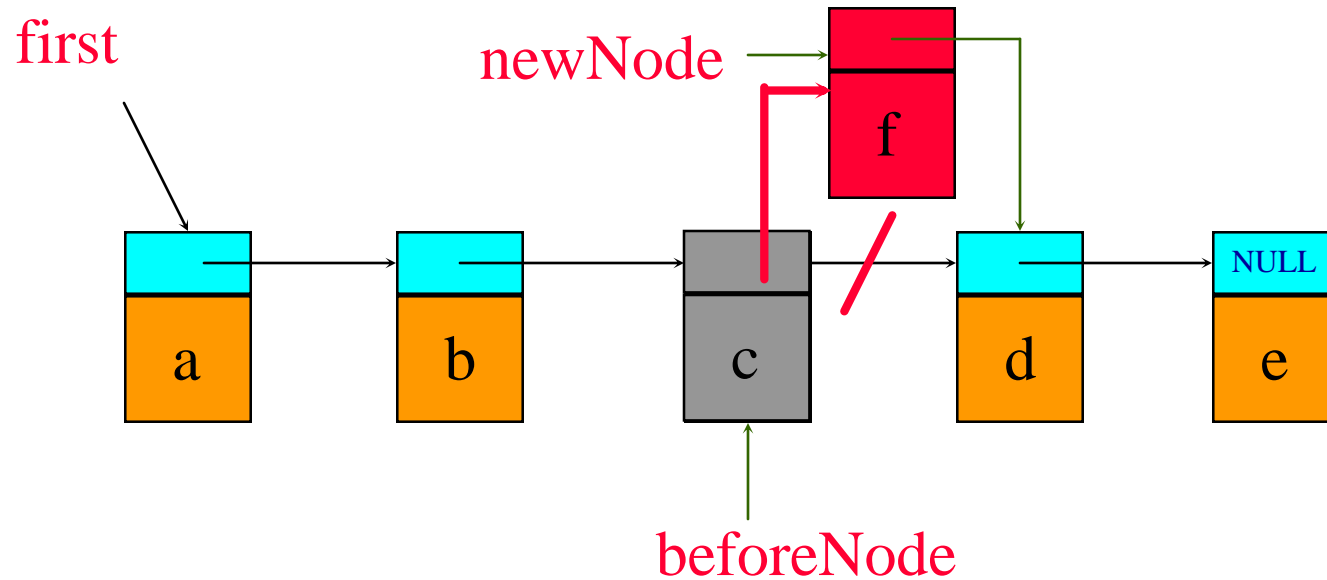


- first find node whose index is 2
- next create a node and set its data and link fields

```
ChainNode<char>* newNode = new ChainNode<char>( 'f',  
                                                beforeNode->link);
```

- finally link **beforeNode** to **newNode**  
`beforeNode->link = newNode;`

# Two-Step Insert(3, 'f')



```
beforeNode = first->link->link;  
beforeNode->link = new ChainNode<char>  
('f', beforeNode->link);
```



# The Template Class Chain

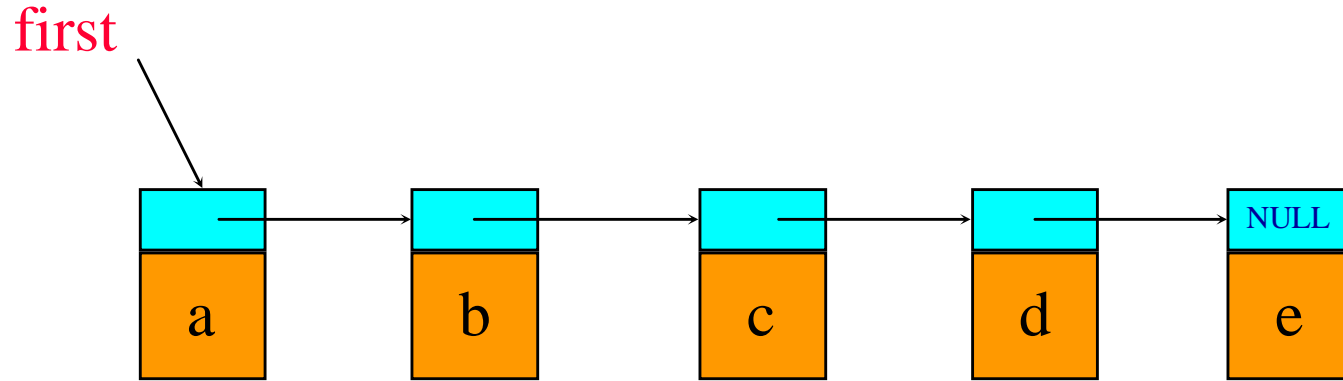


# Chain

- Linear list.
- Each element is stored in a node.
- Nodes are linked together using pointers.



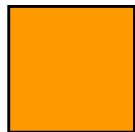
# The Class Chain



Use ChainNode



link (datatype ChainNode<T>\*)



data (datatype T)

# The Template Class Chain

```
template<class T>
class Chain
{
    public:
        Chain() {first = 0;}
            // constructor, empty chain
        ~Chain(); // destructor
        bool IsEmpty() const {return first == 0;}
            // other methods defined here
    private:
        ChainNode<T>* first;
};
```

# The Destructor

```
template<class T>
chain<T>::~~chain()
{ // Chain destructor. Delete all nodes
  // in chain.
  while (first != NULL)
  { // delete first
    ChainNode<T>* next = first->link;
    delete first;
    first = next;
  }
}
```

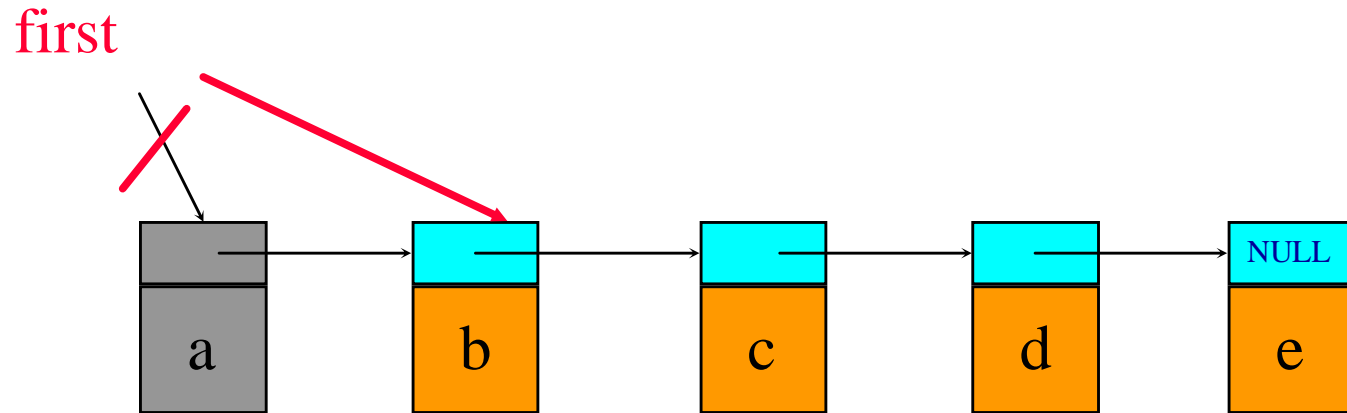
# The Method IndexOf

```
template<class T>
int Chain<T>::IndexOf(const T& theElement) const
{
    // search the chain for theElement
    ChainNode<T>* currentNode = first;
    int index = 0;    // index of currentNode
    while (currentNode != NULL &&
           currentNode->data != theElement)
    {
        // move to next node
        currentNode = currentNode->next;
        index++;
    }
}
```

# The Method IndexOf

```
// make sure we found matching element
if (currentNode == NULL)
    return -1;
else
    return index;
}
```

# Delete An Element



delete(0)

deleteNode = first;

first = first→link;

delete deleteNode;

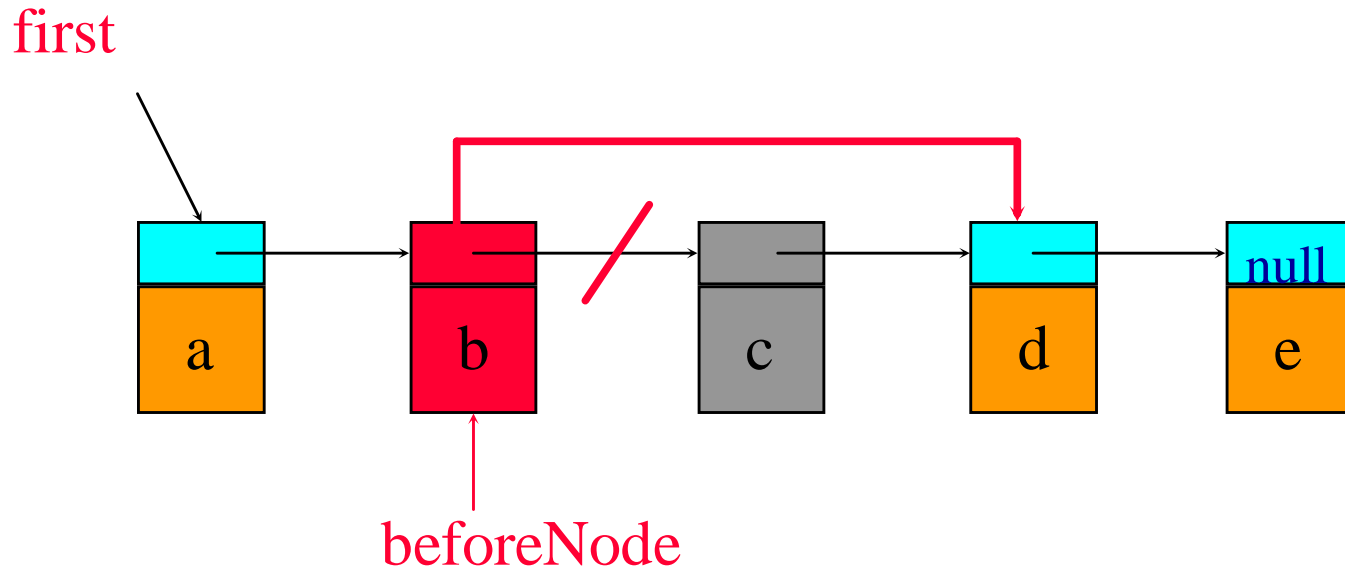


## Delete An Element



```
template<class T>
void Chain<T>::Delete(int theIndex)
{
    if (first == 0)
        throw "Cannot delete from empty chain";
    ChainNode<T>* deleteNode;
    if (theIndex == 0)
        { // remove first node from chain
            deleteNode = first;
            first = first->link;
        }
}
```

## Delete(2)



Find & change pointer in **beforeNode**

**beforeNode-•link = beforeNode-•link-•link;**  
**delete deleteNode;**





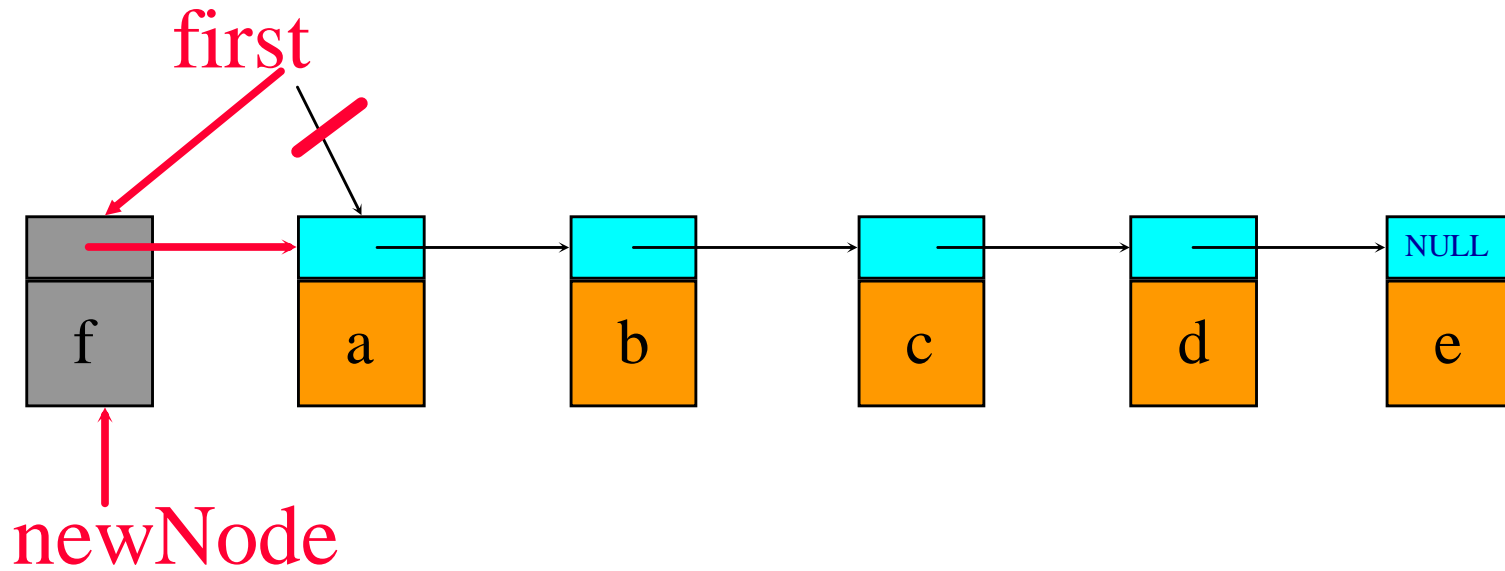
## Delete An Element



else

```
{ // use p to get to beforeNode
  ChainNode<T>* p = first;
  for (int i = 0; i < theIndex - 1; i++)
  {if (p == 0)
    throw "Delete element does not exist";
    p = p->next;}
  deleteNode = p->link;
  p->link = p->link->link;
}
delete deleteNode;
}
```

# One-Step Insert(0, 'f')



```
first = new ChainNode<char>('f', first);
```



## Insert An Element



```
template<class T>
void Chain<T>::Insert(int theIndex,
                     const T& theElement)
{
    if (theIndex < 0)
        throw "Bad insert index";

    if (theIndex == 0)
        // insert at front
        first = new chainNode<T>
            (theElement, first);
```





## Inserting An Element



else

```
{ // find predecessor of new element
  ChainNode<T>* p = first;
  for (int i = 0; i < theIndex - 1; i++)
  {if (p == 0)
    throw "Bad insert index";
    p = p->next;}
  // insert after p
  p->link = new ChainNode<T>
              (theElement, p->link);
}
```

```
}
```

# Iterators & Chain Variants



# Iterators

- An iterator permits you to examine the elements of a data structure one at a time.
- C++ iterators
  - Input iterator
  - Output iterator
  - Forward iterator
  - Bidirectional iterator
  - Reverse iterator

# Forward Iterator

Allows only forward movement through the elements of a data structure.



# Forward Iterator Methods

- `iterator(T* thePosition)`  
Constructs an iterator positioned at specified element
- dereferencing operators `*` and `->`
- Post and pre increment and decrement operators `++`
- Equality testing operators `==` and `!=`

# Bidirectional Iterator

Allows both forward and backward movement through the elements of a data structure.

# Bidirectional Iterator Methods

- `iterator(T* thePosition)`  
Constructs an iterator positioned at specified element
- dereferencing operators `*` and `->`
- Post and pre increment and decrement operators `++` and `--`
- Equality testing operators `==` and `!=`

# Iterator Class

- Assume that a forward iterator class **ChainIterator** is defined within the class **Chain**.
- Assume that methods **Begin()** and **End()** are defined for **Chain**.
  - **Begin()** returns an iterator positioned at element 0 (i.e., leftmost node) of list.
  - **End()** returns an iterator positioned one past last element of list (i.e., NULL or 0).

# Using An Iterator

```
Chain<int>::iterator xHere = x.Begin();  
Chain<int>::iterator xEnd = x.End();  
for (; xHere != xEnd; xHere++)  
    examine( *xHere);
```

VS

```
for (int i = 0; i < x.Size(); i++)  
    examine(x.Get(i));
```

# Merits Of An Iterator

- it is often possible to implement the `++` and `--` operators so that their complexity is less than that of `Get`.
- this is true for a chain
- many data structures do not have a get by index method
- iterators provide a uniform way to sequence through the elements of a data structure

# A Forward Iterator For Chain

```
class ChainIterator {
```

```
public:
```

```
    // some typedefs omitted
```

```
    // constructor comes here
```

```
    // dereferencing operators * & ->, pre and post
```

```
    // increment, and equality testing operators
```

```
    // come here
```

```
private:
```

```
    ChainNode<T> *current;
```

# Constructor

```
ChainIterator(ChainNode<T> * startNode = 0)  
{ current = startNode;}
```



# Dereferencing Operators

```
T& operator*() const  
{ return current->data; }
```

```
T& operator->() const  
{ return &current->data; }
```

# Increment

```
ChainIterator& operator++() // preincrement  
    {current = current->link; return *this;}
```

```
ChainIterator& operator++(int) // postincrement  
{  
    ChainIterator old = *this;  
    current = current->link;  
    return old;  
}
```

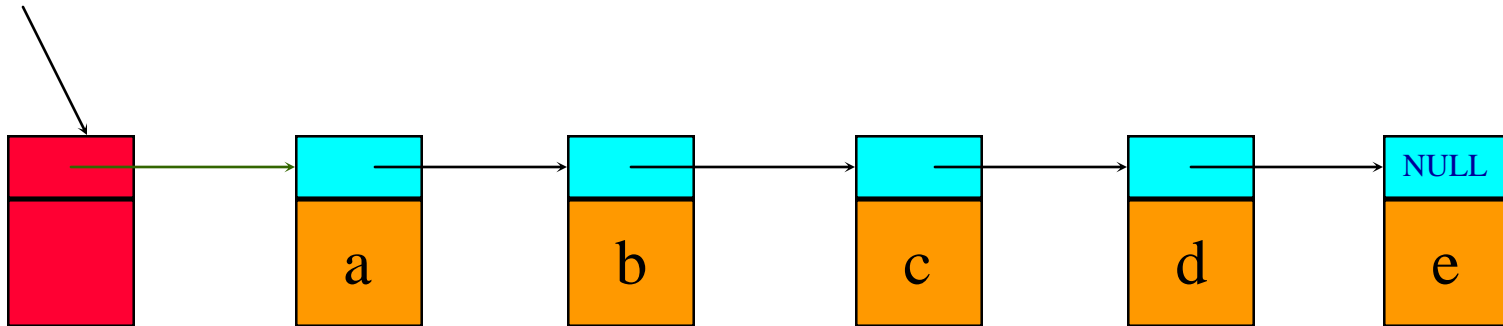
# Equality Testing

```
bool operator!=(const ChainIterator right) const  
{ return current != right.current; }
```

```
bool operator==(const ChainIterator right) const  
{ return current == right.current; }
```

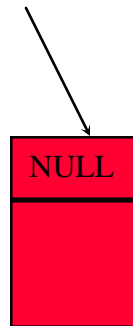
# Chain With Header Node

headerNode



# Empty Chain With Header Node

headerNode

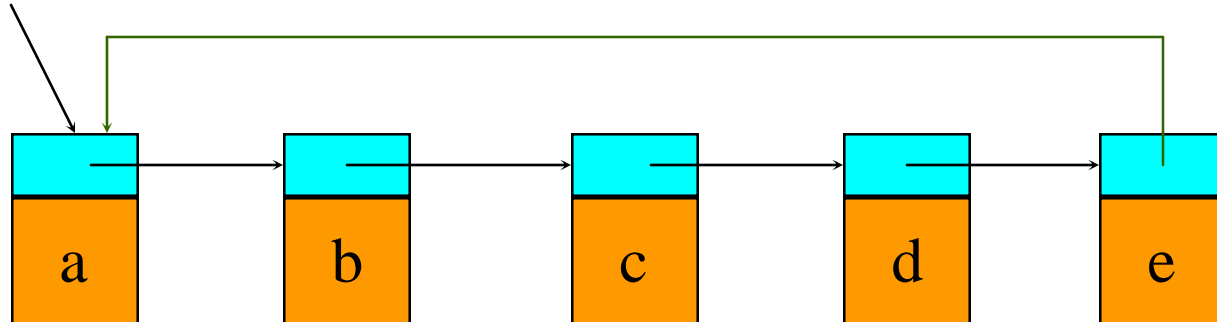




# Circular List



firstNode



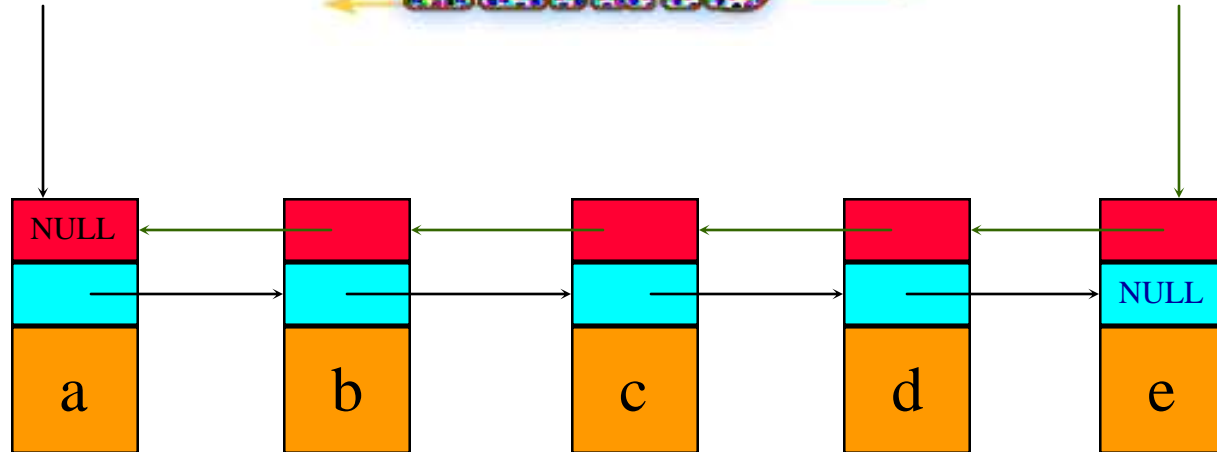


# Doubly Linked List



firstNode

lastNode

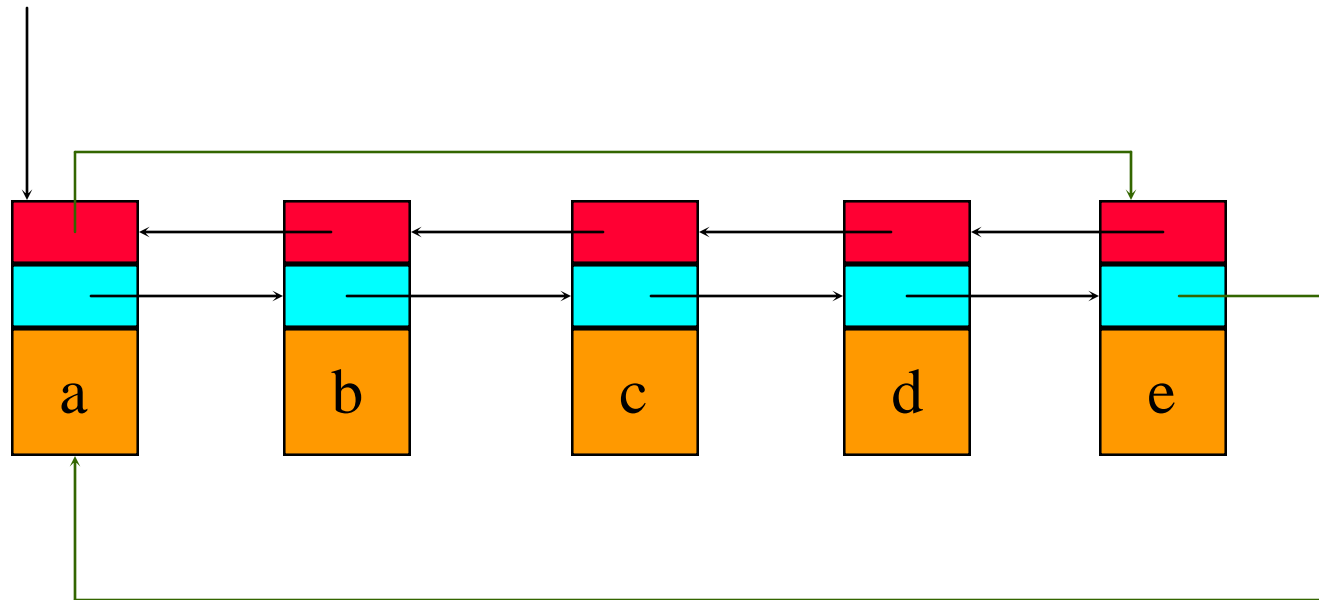




# Doubly Linked Circular List



firstNode



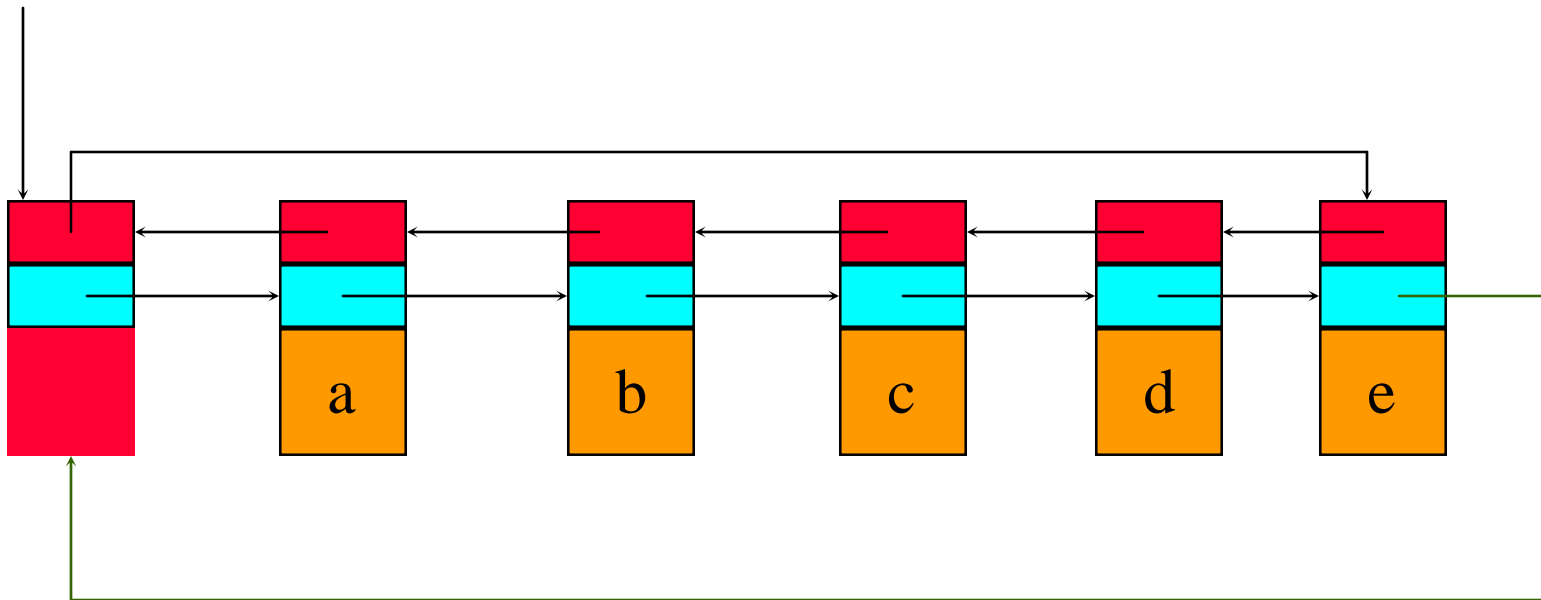




# Doubly Linked Circular List With Header Node



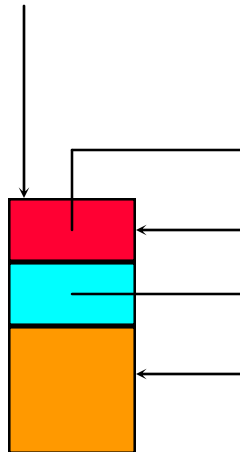
headerNode



# Empty Doubly Linked Circular List With Header Node



headerNode



# The STL Class **list**

- Linked implementation of a linear list.
- Doubly linked circular list with header node.
- Has many more methods than our **Chain**.
- Similar names and signatures.

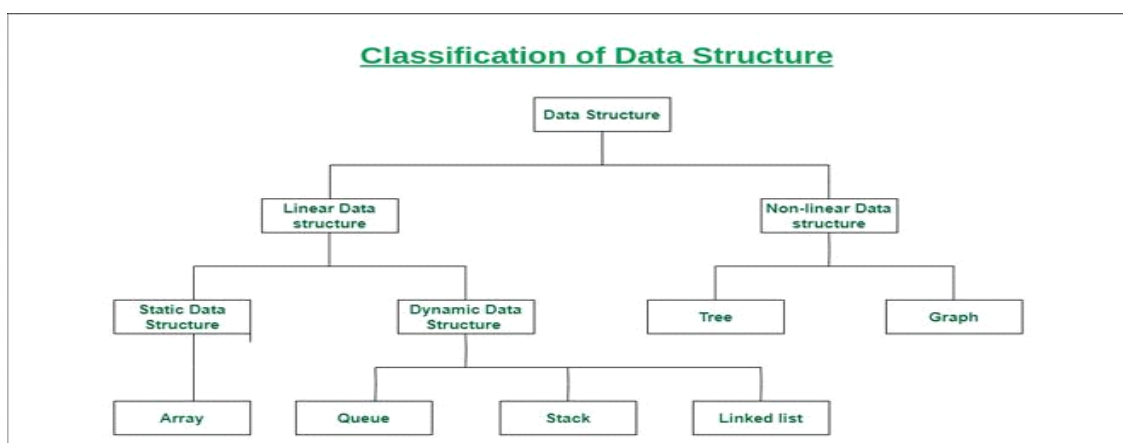
## Introduction: Basic Terminology, Elementary Data Organization

- A **good program** is defined as a program that runs correctly, easy to read and understand, easy to debug and easy to modify.
- A program is said to be efficient when it executes in minimum time and with minimum memory space.
- In order to write efficient programs, we need to apply certain **data management concepts**.
- Data structure is a crucial part of data management.
- A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.
- **Data**: Data are simply values or sets of values.
- **Data items**: Data items refers to a single unit of values. Data items that are divided into sub-items are called Group items. Ex: An Employee Name may be divided into three subitems- first name, middle name, and last name.
- Data items that are not able to divide into sub-items are called **Elementary items**. Ex: SSN
- **Entity**: An entity is something that has certain attributes or properties which may be assigned values. The values may be either numeric or non-numeric. Ex: Attributes- Names, Age, Sex, SSN Values- Rohland Gail, 34, F, 134-34-5533 Entities with similar attributes form an entity set. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute.
- The term “**information**” is sometimes used for data with given attributes, of, in other words meaningful or processed data.
- **Field** is a single elementary unit of information representing an attribute of an entity.
- **Record** is the collection of field values of a given entity.
- **File** is the collection of records of the entities in a given entity set.
- Each record in a file may contain many field items but the value in a certain field may uniquely determine the record in the file. Such a field K is called a **primary key**.
- Records may also be classified according to length. A file can have fixed-length records or variable-length records.
- In **fixed-length records**, all the records contain the same data items with the same amount of space assigned to each data item.
- In **variable-length records** file records may contain different lengths. Example: Student records have variable lengths, since different students take different numbers of courses. Variable-length records have a minimum and a maximum length.

### What do you mean by the term “Data Structure”?

- Organizing the data in memory.
- A data structure is a way of organizing the data so that it can be used efficiently. Here, we have used the word efficiently, which in terms of both the space and time.
- It is a set of algorithms that we can use in any programming language to structure the data in the memory.

### Classification of Data Structure:



#### • Linear data structure:

Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure. Examples of linear data structures are array, stack, queue, linked list, etc.

**Static data structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure. An example of this data structure is an array.

**Dynamic data structure:** In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code. Examples of this data structure are queue, stack, etc.

- **Non-linear data structure:**

Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only. Examples of non-linear data structures are trees and graphs.

- **The major or the common operations that can be performed on the data structures are:**

- Searching: We can search for any element in a data structure.
- Sorting: We can sort the elements of a data structure either in an ascending or descending order.
- Insertion: We can also insert the new element in a data structure.
- Updation: We can also update the element, i.e., we can replace the element with another element.
- Deletion: We can also perform the delete operation to remove the element from the data structure.

- **Advantages of Data structures**

The following are the advantages of a data structure:

- Efficiency: If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.
- Reusability: The data structure provides reusability means that multiple client programs can use the data structure.
- Abstraction: The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

- **An abstract data type (ADT)** is the way we look at a data structure, focusing on what it does and ignoring how it does its job. For example, stacks and queues are perfect examples of an ADT. We can implement both these ADTs using an array or a linked list. This demonstrates the 'abstract' nature of stacks and queues. They are not concerned about how they work.

## Arrays: Definition, Declaration, Initialization & Processing of Single and Multidimensional Arrays

### Array Data Structure

#### Definition

- An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.
- They exist in both single dimension and multiple dimensions.

#### Basic terminologies of the array:

- Array Index: In an array, elements are identified by their indexes. Array index starts from 0.
- Array element: Elements are items stored in an array and can be accessed by their index.
- Array Length: The length of an array is determined by the number of elements it can contain.

#### Syntax

Creating an array in C language –Arrays are declared using the following syntax:

```
type name[size];
```

For example, if we write, `int marks[10];`

#### Types of arrays:

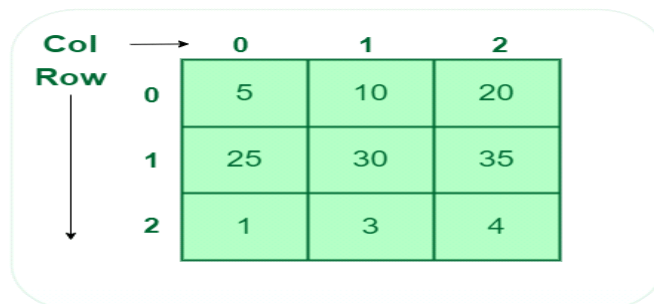
There are following types of arrays:

- One-dimensional array (1-D arrays): You can imagine a 1d array as a row, where elements are stored one after another.



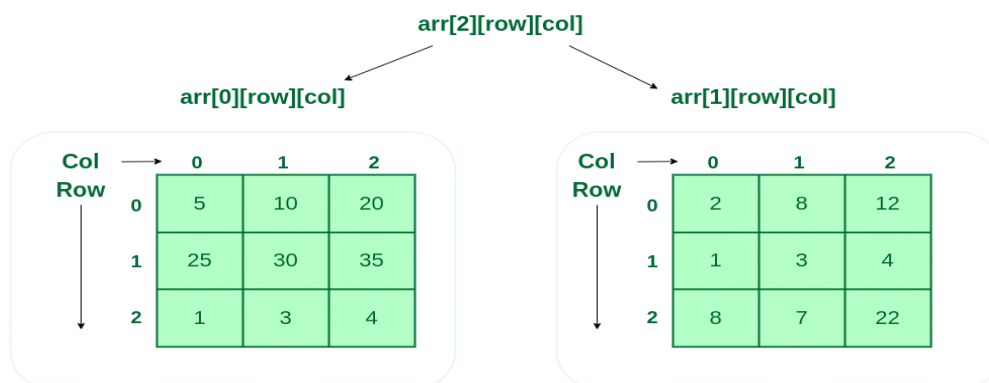
#### 1D array

- Two-dimensional array: 2-D Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.



#### 2D array

- Three-dimensional array: A 3-D Multidimensional array contains three dimensions, so it can be considered an array of two-dimensional arrays.



#### 3D array

## Declaration, Initialization & Processing of Single and Multidimensional Arrays

### One Dimensional Array

#### Declaration:

Syntax: `data_type array_name[size];`

`data_type` represents the type of elements present in the array. `array_name` represents the name of the array.

Size represents the number of elements that can be stored in the array.

Example: `int age[100]; float sal[15]; char grade[20];`

Here `age` is an integer type array, which can store 100 elements of integer type. The array `sal` is floating type array of size 15, can hold float values. `Grade` is a character type array which holds 20 characters.

#### Initialization:

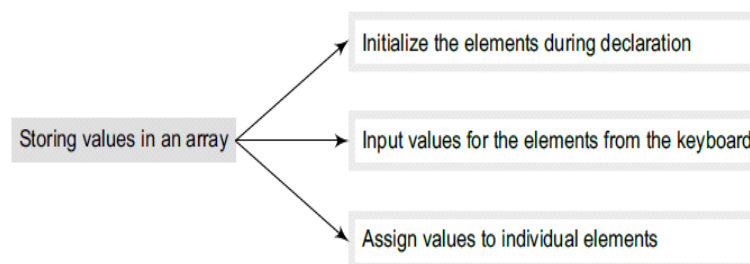
Syntax:

`data_type array_name[size]={value1, value2,.....valueN};`

`Value1, value2, valueN` are the constant values known as initializers, which are assigned to the array elements one after another.

Example: `int marks[5]={10,2,0,23,4};`

The values of the array elements after this initialization are: `marks[0]=10, marks[1]=2, marks[2]=0, marks[3]=23, marks[4]=4`



#### Note:-

In 1-D arrays it is optional to specify the size of the array. If size is omitted during initialization, then the compiler assumes the size of array equal to the number of initializers. Example: `int marks []={10,2,0,23,4};` Here the size of array `marks` is initialized to 5.

We can't copy the elements of one array to another array by simply assigning it. Example:

```
int a[5]={9,8,7,6,5}; int b[5];
```

```
b=a; //not valid
```

we have to copy all the elements by using for loop.

```
for(a=i; i<5; i++) b[i]=a[i];
```

#### Processing:

For processing arrays we mostly use for loop. The total no. of passes is equal to the no. of elements present in the array and in each pass one element is processed.

Example:

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

```
main()
```

```
{  
  int a[3],i;
```

```
  for(i=0;i<=2;i++) //Reading the array values
```

```
  {  
    printf("enter the elements");
```

```
    scanf("%d",&a[i]);
```

```
  }  
  for(i=0;i<=2;i++) //display the array values
```

```
  { printf("%d",a[i]);
```

```
    printf("\n");
```

```
  } }
```

#### Two Dimensional Arrays-

Arrays that we have considered up to now are one dimensional array, a single line of elements. Often data come naturally in the form of a table, e.g. spreadsheet, which need a two-dimensional array.

**Declaration:**

Syntax: `data_type array_name[rowsize][columnsize];`

Rowsize specifies the no. of rows, Columnsize specifies the no. of columns.

Example: `int a[4][5];`

This is a 2-D array of 4 rows and 5 columns. Here the first element of the array is `a[0][0]` and last element of the array is `a[3][4]` and total no. of elements is  $4 \times 5 = 20$ .

**Initialization:**

2-D arrays can be initialized in a way similar to 1-D arrays.

Example: `int m[4][3] = {1,2,3,4,5,6,7,8,9,10,11,12};`

**Note:**

In 2-D arrays it is optional to specify the first dimension but the second dimension should always be present.

Example: `int m[][3] = {`

`{1,10},`

`{2,20,200},`

`{3}, {4,40,400} };`

Here the first dimension is taken 4 since there are 4 rows in the initialization list. A 2-D array is known as matrix.

**Processing:**

For processing of 2-D arrays we need two nested for loops. The outer loop indicates the rows and the inner loop indicates the columns.

Example:

`int a[4][5];`

Reading values in a

`for(i=0;i<4;i++)`

`for(j=0;j<5;j++)`

`scanf("%d",&a[i][j]);`

Displaying values of a

`for(i=0;i<4;i++)`

`for(j=0;j<5;j++)`

`printf("%d",a[i][j]);`

This program reads and displays 3 elements of integer type.

**Declaration of Three-Dimensional Array:**

We can declare a 3D array with x 2D arrays each having y rows and z columns using the syntax shown below.

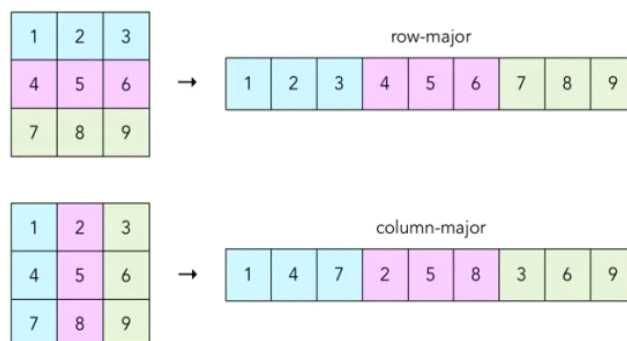
Syntax:

`data_type array_name[x][y][z];`

- `data_type`: Type of data to be stored in each element.
- `array_name`: name of the array
- `x`: Number of 2D arrays.
- `y`: Number of rows in each 2D array.
- `z`: Number of columns in each 2D array.

**Representation of Arrays: Row Major Order, and Column Major Order**

- When it comes to organizing and accessing elements in a multi-dimensional array, two prevalent methods are Row Major Order and Column Major Order.



These approaches define how elements are stored in memory and impact the efficiency of data access in computing.



## Row Major Order

Row major ordering assigns successive elements, moving across the rows and then down the next row, to successive memory locations. In simple language, the elements of an array are stored in a Row-Wise fashion.

To find the address of the element using row-major order uses the following formula:

$$\text{Address of } A[I][J] = B + W * ((I - LR) * N + (J - LC))$$

I = Row Subset of an element whose address to be found,

J = Column Subset of an element whose address to be found,

B = Base address,

W = Storage size of one element store in an array(in byte),

LR = Lower Limit of row/start row index of the matrix(If not given assume it as zero),

LC = Lower Limit of column/start column index of the matrix(If not given assume it as zero),

N = Number of column given in the matrix.

## How to find address using Row Major Order?

**Q-**Given an array, arr[1.....10][1.....15] with base value 100 and the size of each element is 1 Byte in memory. Find the address of arr[8][6] with the help of row-major order.

**Solution:**Given: Base address B = 100

Storage size of one element store in any array W = 1 Bytes

Row Subset of an element whose address to be found I = 8

Column Subset of an element whose address to be found J = 6

Lower Limit of row/start row index of matrix LR = 1

Lower Limit of column/start column index of matrix = 1

Number of columns given in the matrix N = Upper Bound – Lower Bound + 1

$$= 15 - 1 + 1$$

$$= 15$$

Formula:

$$\text{Address of } A[I][J] = B + W * ((I - LR) * N + (J - LC))$$

$$\text{Address of } A[8][6] = 100 + 1 * ((8 - 1) * 15 + (6 - 1))$$

$$= 100 + 1 * ((7) * 15 + (5))$$

$$= 100 + 1 * (110)$$

$$\text{Address of } A[I][J] = 210$$

## Column Major Order

If elements of an array are stored in a column-major fashion means moving across the column and then to the next column then it's in column-major order.

To find the address of the element using column-major order use the following formula:

$$\text{Address of } A[I][J] = B + W * ((J - LC) * M + (I - LR))$$

I = Row Subset of an element whose address to be found,

J = Column Subset of an element whose address to be found,

B = Base address,

W = Storage size of one element store in any array(in byte),

LR = Lower Limit of row/start row index of matrix(If not given assume it as zero),

LC = Lower Limit of column/start column index of matrix(If not given assume it as zero),

M = Number of rows given in the matrix.

## How to find address using Column Major Order?

**Q-**Given an array arr[1.....10][1.....15] with a base value of 100 and the size of each element is 1 Byte in memory find the address of arr[8][6] with the help of column-major order.

**Solution:**Given:

Base address B = 100

Storage size of one element store in any array W = 1 Bytes

Row Subset of an element whose address to be found I = 8

Column Subset of an element whose address to be found J = 6

Lower Limit of row/start row index of matrix LR = 1

Lower Limit of column/start column index of matrix = 1

Number of Rows given in the matrix M = Upper Bound – Lower Bound + 1

$$= 10 - 1 + 1$$

$$= 10$$

Formula: used

$$\text{Address of } A[I][J] = B + W * ((J - LC) * M + (I - LR))$$

$$\begin{aligned} \text{Address of } A[8][6] &= 100 + 1 * ((6 - 1) * 10 + (8 - 1)) \\ &= 100 + 1 * ((5) * 10 + (7)) \\ &= 100 + 1 * (57) \end{aligned}$$

$$\text{Address of } A[I][J] = 157$$

## Operations supported by an array.

### 1. Traverse and display

Write a program to read and display n numbers using an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int i, n, arr[20];
    clrscr();
    printf("\n Enter the no of elements in the array : ");
    scanf("%d", &n);
    for(i=0;i<n;i++)
    {
        printf("\n arr[%d] = ", i);
        scanf("%d",&arr[i]);
    }
    printf("\n The array elements are ");
    for(i=0;i<n;i++)
        printf("\t %d", arr[i]);
    return 0;
}
```

Output

Enter the number of elements in the array : 5

arr[0] = 1

arr[1] = 2

arr[2] = 3

arr[3] = 4

arr[4] = 5

The array elements are 1 2 3 4 5

### 2. Insertion

Write a program to insert a number at a given location in an array.

```
#include <stdio.h>
int main()
{
    int i, n, num, pos, arr[10];
    clrscr();
    printf("\n Enter the no of elements in the array : ");
    scanf("%d", &n);
    for(i=0;i<n;i++)
    {
        printf("\n arr[%d] = ", i);
        scanf("%d", &arr[i]);
    }
    printf("\n Enter the number to be inserted : ");
    scanf("%d", &num);
    printf("\n Enter the position at which the number has to be added : ");
    scanf("%d", &pos);
    for(i=n-1;i>=pos;i--)
        arr[i+1] = arr[i];
```

```

arr[pos] = num;
n = n+1;
printf("\n The array after insertion of %d is : ", num);
for(i=0;i<n;i++)
printf("\n arr[%d] = %d", i, arr[i]);
return 0;
}

```

Output

Enter the number of elements in the array : 5

arr[0] = 1

arr[1] = 2

arr[2] = 3

arr[3] = 4

arr[4] = 5

Enter the number to be inserted : 0

Enter the position at which the number has to be added : 3

The array after insertion of 0 is :

arr[0] = 1

arr[1] = 2

arr[2] = 3

arr[3] = 0

arr[4] = 4

arr[5] = 5

### 3.Deletion

Write a program to delete a number from a given location in an array.

```

#include <stdio.h>
int main()
{
int i, n, pos, arr[10];
clrscr();
printf("\n Enter the no of elements in the array : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{printf("\n arr[%d] = ", i);
scanf("%d", &arr[i]);
}
printf("\nEnter the position from which the number has to be deleted : ");
scanf("%d", &pos);
for(i=pos; i<n-1;i++)
arr[i] = arr[i+1];
n--;
printf("\n The array after deletion is : ");
for(i=0;i<n;i++)
printf("\n arr[%d] = %d", i, arr[i]);
return 0;
}

```

Output

Enter the number of elements in the array : 5

arr[0] = 1

arr[1] = 2

arr[2] = 3

arr[3] = 4

arr[4] = 5

Enter the position from which the number has to be deleted : 3

The array after deletion is :

arr[0] = 1

arr[1] = 2

arr[2] = 3

arr[3] = 5

#### 4.Search &Update

```
#include<stdio.h>
int main()
{ int i,t,a[10],n,m,s,j=0,b[10];
  printf("\nEnter the Limit:");
  scanf("%d",&n);
  printf("\nEnter the Values:");
  for(i=0;i<n;i++)
  {   scanf("%d",&a[i]);
  }
  printf("\nGiven values are:");
  for(i=0;i<n;i++)
  {   printf("a[%d]=%d",i,a[i]);
  }
  printf("\nEnter the position to be update:");
  scanf("%d",&t);
  printf("\nEnter the value to be update:");
  scanf("%d",&s);
  for(i=0;i<n;i++)
  {
    if(i==t)
    {   a[i]=s;
    }
  }
  printf("\nUpdated value is:");
  for(i=0;i<n;i++)
  {   printf("\na[%d]=%d",i,a[i]);
  }
  return 0;
}
```

##### Output

```
Enter the limit:5
Enter the values:1 2 3 4 5
Given values are:
a[0]=1
a[1]=2
a[2]=3
a[3]=4
a[4]=5
Enter the position to be update:3
Enter the value to be update:5
Inserted value is:
a[0]=1
a[1]=2
a[2]=3
a[3]=5
a[4]=4
a[5]=5
```

#### Basic Operations on 2D array:

##### 1.Write a program to transpose a 3 x 3 matrix.

```
#include <stdio.h>
int main()
{
  int i, j, mat[3][3], transposed_mat[3][3];
  clrscr();
  printf("\n Enter the elements of the matrix ");
  for(i=0;i<3;i++)
  {
```

```

for(j=0;j<3;j++)
{
scanf("%d", &mat[i][j]);
}
}
printf("\n The elements of the matrix are ");
for(i=0;i<3;i++)
{
printf("\n");
for(j=0;j<3;j++)
printf("\t %d", mat[i][j]);
}
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
transposed_mat[i][j] = mat[j][i];
}
printf("\n The elements of the transposed matrix are ");
for(i=0;i<3;i++)
{
printf("\n");
for(j=0;j<3;j++)
printf("\t %d",transposed_mat[i][j]);}
return 0;
}

```

Output

Enter the elements of the matrix

1 2 3 4 5 6 7 8 9

The elements of the matrix are

1 2 3

4 5 6

7 8 9

The elements of the transposed matrix are

1 4 7

2 5 8

3 6 9

**2. Write a program to input two m x n matrices and then calculate the sum of their corresponding elements and store it in a third m x n matrix.**

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

```
int main()
```

```
{
```

```
int i, j;
```

```
int rows1, cols1, rows2, cols2, rows_sum, cols_sum;
```

```
int mat1[5][5], mat2[5][5], sum[5][5];
```

```
clrscr();
```

```
printf("\n Enter the number of rows in the first matrix : ");
```

```
scanf("%d",&rows1);
```

```
printf("\n Enter the number of columns in the first matrix : ");
```

```
scanf("%d",&cols1);
```

```
printf("\n Enter the number of rows in the second matrix : ");
```

```
scanf("%d",&rows2);
```

```
printf("\n Enter the number of columns in the second matrix : ");
```

```
scanf("%d",&cols2);
```

```
if(rows1 != rows2 || cols1 != cols2)
```

```
{
```

```
printf("\n Number of rows and columns of both matrices must be equal");
```

```
exit(0);
```

```
}
```

```
rows_sum = rows1;
```

```
cols_sum = cols1;
```

```

printf("\n Enter the elements of the first matrix ");
for(i=0;i<rows1;i++)
{
for(j=0;j<cols1;j++)
{
scanf("%d",&mat1[i][j]);
}
}
printf("\n Enter the elements of the second matrix ");
for(i=0;i<rows2;i++)
{
for(j=0;j<cols2;j++)
{
scanf("%d",&mat2[i][j]);
}
}
for(i=0;i<rows_sum;i++)
{
for(j=0;j<cols_sum;j++)
sum[i][j] = mat1[i][j] + mat2[i][j];
}
printf("\n The elements of the resultant matrix are ");
for(i=0;i<rows_sum;i++)
{
printf("\n");
for(j=0;j<cols_sum;j++)
printf("\t %d", sum[i][j]);
}
return 0;
}

```

#### Output

```

Enter the number of rows in the first matrix: 2
Enter the number of columns in the first matrix: 2
Enter the number of rows in the second matrix: 2
Enter the number of columns in the second matrix: 2
Enter the elements of the first matrix
1 2 3 4
Enter the elements of the second matrix
5 6 7 8
The elements of the resultant matrix are
6 8
10 12

```

### 3. Write a program to multiply two m x n matrices.

```

#include <stdio.h>
int main()
{
int i, j, k;
int rows1, cols1, rows2, cols2, res_rows, res_cols;
int mat1[5][5], mat2[5][5], res[5][5];
clrscr();
printf("\n Enter the number of rows in the first matrix : ");
scanf("%d",&rows1);
printf("\n Enter the number of columns in the first matrix : ");
scanf("%d",&cols1);
printf("\n Enter the number of rows in the second matrix : ");
scanf("%d",&rows2);
printf("\n Enter the number of columns in the second matrix : ");
scanf("%d",&cols2);
if(cols1 != rows2)

```

```

{
printf("\n The number of columns in the first matrix must be equal
to the number of rows in the second matrix");
exit();
}
res_rows = rows1;
res_cols = cols2;
printf("\n Enter the elements of the first matrix ");
for(i=0;i<rows1;i++)
{
for(j=0;j<cols1;j++)
{
scanf("%d",&mat1[i][j]);
}
}
printf("\n Enter the elements of the second matrix ");
for(i=0;i<rows2;i++)
{
for(j=0;j<cols2;j++)
{
scanf("%d",&mat2[i][j]);
}
}
for(i=0;i<res_rows;i++)
{
for(j=0;j<res_cols;j++)
Arrays 103
{
res[i][j]=0;
for(k=0; k<res_cols;k++)
res[i][j] += mat1[i][k] * mat2[k][j];
}
}
printf("\n The elements of the product matrix are ");
for(i=0;i<res_rows;i++)
{
printf("\n");
for(j=0;j<res_cols;j++)
printf("\t %d",res[i][j]);
}
return 0;
}

```

#### Output

```

Enter the number of rows in the first matrix: 2
Enter the number of columns in the first matrix: 2
Enter the number of rows in the second matrix: 2
Enter the number of columns in the second matrix: 2
Enter the elements of the first matrix
1 2 3 4
Enter the elements of the second matrix
5 6 7 8
The elements of the product matrix are
19 22
43 50

```

#### Pointer and 2D Array:

**Write a program to read and display a 3 x 3 matrix.**

```

#include <stdio.h>
#include <conio.h>
void display(int (*)[3]);
int main()

```

```

{
int i, j, mat[3][3];
clrscr();
printf("\n Enter the elements of the matrix");
for(i=0;i<3;i++)
{
for(j = 0; j < 3; j++)
{
scanf("%d", &mat[i][j]);
}
}
display(mat);
return 0;
}
void display(int (*mat)[3])
{
int i, j;
printf("\n The elements of the matrix are");
for(i = 0; i < 3; i++)
{
printf("\n");
for(j=0;j<3;j++)
printf("\t %d",*(mat + i)+j));
}
}
}

```

Output

Enter the elements of the matrix

1 2 3 4 5 6 7 8 9

The elements of the matrix are

1 2 3

4 5 6

7 8 9

### Pointer and 3D Array

**Write a program which illustrates the use of a pointer to a three-dimensional array.**

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

```

int main()
{
int i,j,k;
int arr[2][2][2];
int (*parr)[2][2]= arr;
clrscr();
printf("\n Enter the elements of a 2 \ 2 \ 2 array: ");
for(i = 0; i < 2; i++)
{
for(j = 0; j < 2; j++)
{
for(k = 0; k < 2; k++)
scanf("%d", &arr[i][j][k]);
}
}
printf("\n The elements of the 2 \ 2 \ 2 array are: ");
for(i = 0; i < 2; i++)
{
for(j = 0; j < 2; j++)
{
for(k = 0; k < 2; k++)
printf("%d", *((*(parr+i)+j)+k));
}
}
return 0;
}

```



```
}
```

Output

Enter the elements of a 2 \ 2 \ 2 array: 1 2 3 4 5 6 7 8

The elements of the 2 \ 2 \ 2 array are: 1 2 3 4 5 6 7 8

**Note** In the printf statement, you could also have used `* ( * ( * ( a r + i ) + j + ) + k )` instead of `*(*(*(parr+i)+j)+k)`.

### **Advantages of using Arrays:**

- Arrays allow random access to elements. This makes accessing elements by position faster.
- Arrays have better cache locality which makes a pretty big difference in performance.
- Arrays represent multiple data items of the same type using a single name.
- Arrays store multiple data of similar types with the same name.
- Array data structures are used to implement the other data structures like linked lists, stacks, queues, trees, graphs, etc.

### **Disadvantages of Array:**

- As arrays have a fixed size, once the memory is allocated to them, it cannot be increased or decreased, making it impossible to store extra data if required. An array of fixed size is referred to as a static array.
- Allocating less memory than required to an array leads to loss of data.
- An array is homogeneous in nature so, a single array cannot store values of different data types.
- Arrays store data in contiguous memory locations, which makes deletion and insertion very difficult to implement. This problem is overcome by implementing linked lists, which allow elements to be accessed sequentially.

### **Application of arrays**

- They are used in the implementation of other data structures such as array lists, heaps, hash tables, vectors, and matrices.
- Database records are usually implemented as arrays.
- It is used for different sorting algorithms such as bubble sort insertion sort, merge sort, and quick sort.
- It is used for implementing matrices.
- Graphs are also implemented as arrays in the form of an adjacency matrix.

## [T-6] Sparse Matrices and their representations

- A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values.
- If most of the elements of the matrix have 0 value, then it is called a sparse matrix.

### Why to use Sparse Matrix instead of simple matrix?

- Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements.

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with triples- (Row, Column, value).

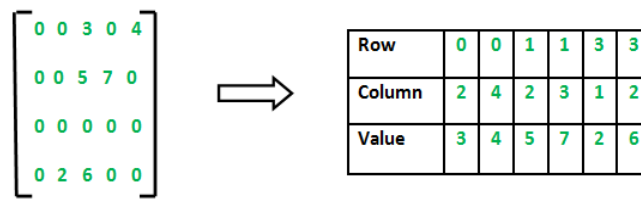
Sparse Matrix Representations can be done in many ways following are two common representations:

- Array representation
- Linked list representation\*

Method 1: Using Arrays:

2D array is used to represent a sparse matrix in which there are three rows named as

- Row: Index of row, where non-zero element is located
- Column: Index of column, where non-zero element is located
- Value: Value of the non-zero element located at index – (row, column)



### Implementation in C

```
// C program for Sparse Matrix Representation // using Array
```

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

```
int main()
```

```
{ // Assume 4x5 sparse matrix
```

```
int sparseMatrix[4][5] =
```

```
{
```

```
{0, 0, 3, 0, 4},
```

```
{0, 0, 5, 7, 0},
```

```
{0, 0, 0, 0, 0},
```

```
{0, 2, 6, 0, 0}
```

```
};
```

```
int size = 0;
```

```
for (int i = 0; i < 4; i++)
```

```
for (int j = 0; j < 5; j++)
```

```
if (sparseMatrix[i][j] != 0)
```

```
size++;
```

```
// number of columns in compactMatrix (size) must be equal to number of non - zero elements in sparseMatrix
```

```
int compactMatrix[3][size];
```

```
// Making of new matrix
```

```
int k = 0;
```

```
for (int i = 0; i < 4; i++)
```

```
for (int j = 0; j < 5; j++)
```

```
if (sparseMatrix[i][j] != 0)
```

```
{
```

```
compactMatrix[0][k] = i;
```

```

        compactMatrix[1][k] = j;
        compactMatrix[2][k] = sparseMatrix[i][j];
        k++;
    }

    for (int i=0; i<3; i++)
    {
        for (int j=0; j<size; j++)
            printf("%d ", compactMatrix[i][j]);

        printf("\n");
    }
    return 0;
}

```

Output

```

0 0 1 1 3 3
2 4 2 3 1 2
3 4 5 7 2 6

```

```

1  #include<stdio.h>
2  #include<stdlib.h>
3  typedef struct node
4  {
5      int data;
6      struct node *next;
7  }node;
8
9  void insert();
10 void reverse();
11 void disp();
12 struct node *start=NULL;
13

```

```

13
14 void main()
15 {
16     int ch;
17     while(1)
18     {
19         printf("\n---Options Are---");
20         printf("\n1-Insertion");
21         printf("\n2-Revers");
22         printf("\n3-Display");
23         printf("\n4-Exit");
24         printf("\nEnter your choice: ");
25         scanf("%d",&ch);
26         switch(ch)
27         {
28             case 1: insert();
29                     break;
30             case 2: reverse();
31                     break;
32             case 3: disp();
33                     break;
34             case 4: exit(0);
35             default: printf("\n Wrong Choice");
36         }
37     }
38 }
39

```

```

39
40 void insert()
41 {
42     node *ptr;
43     ptr = (node*)malloc(sizeof(node));
44     printf("\nEnter the value: ");
45     scanf("%d",&ptr->data);
46     ptr->next=NULL;
47     if(start==NULL)
48         start=ptr;
49     else
50     {
51         node *temp;
52         temp = start;
53         while(temp->next!=NULL)
54             temp = temp->next;
55         temp->next=ptr;
56     }
57 }
58

```

```

58
59 void reverse()
60 {
61     struct node *temp1,*temp2;
62     temp1=start;
63     start=start->next;
64     temp1->next=NULL;
65     temp2=start;
66     while(start!=NULL)
67     {
68
69         start=start->next;
70         temp2->next=temp1;
71         temp1=temp2;
72         temp2=start;
73     }
74     start=temp1;
75 }
76
80
81
82 void disp()
83 {
84     struct node *temp;
85     temp = start;
86     if(temp==NULL)
87         printf("\nList is empty");
88     else
89     {
90         printf("\nYou have entered following data");
91         while(temp)
92         {
93             printf(" %d ",temp->data);
94             temp=temp->next;
95         }
96     }
97 }

```

### 3.2.3 Applications of Linear Arrays

Linear arrays are one of the primitive data structures. These are used to represent all sorts of lists. In addition, these are used to implement other data structures such as *stacks*, *queues*, *heaps*, etc..

### 3.2.4 Limitations of Linear Arrays

The main limitations of linear arrays are

1. The prior knowledge of number of elements in the linear array is necessary.
2. These are static structures. Static in the sense that whether memory is allocated at compilation time or runtime, their memory used by them cannot be reduced or extended.
3. Since the elements of these arrays are stored in consecutive locations, the insertions and deletions in these arrays are time consuming. This is because of moving down or up to create a space of new element or to occupy the space vacated by the deleted element.

## 3.3 TWO-DIMENSIONAL ARRAYS

A two-dimensional array is a list of a finite number  $m \times n$  of homogeneous data elements such that

- The elements of the array are referenced by two index sets consisting of  $m$  and  $n$  consecutive integer numbers.
- The elements of the array are stored in consecutive memory locations.

The size of the two-dimensional array is denoted by  $m \times n$  and pronounced as  $m$  by  $n$ .

Suppose  $b$  is the name of the two-dimensional array, then its element in the  $i$ th row and  $j$ th column is denoted as

$b_{ij}$	in mathematical notation
$b(i, j)$	in BASIC and FORTRAN languages
$b[i, j]$	in Pascal language
$b[i][j]$	in C/C++ and Java languages

Two-dimensional arrays are called *matrices* in mathematics and *tables* in business applications. A two-dimensional  $m \times n$  array is pictured as a rectangular pattern of  $m$  rows and  $n$  columns, where the element  $a_{ij}$  appears in row  $i$  and column  $j$ . Figure 3.6 pictures two-dimensional array  $a$  of size  $3 \times 3$ .

	Col #0	Col #1	Col #2
Row #0	$a_{00}$	$a_{01}$	$a_{02}$
Row #1	$a_{10}$	$a_{11}$	$a_{12}$
Row #2	$a_{20}$	$a_{21}$	$a_{22}$

**Figure 3.6** Picturing a two-dimensional array  $a$

### 3.3.1 Representing Two-dimensional Array in Memory

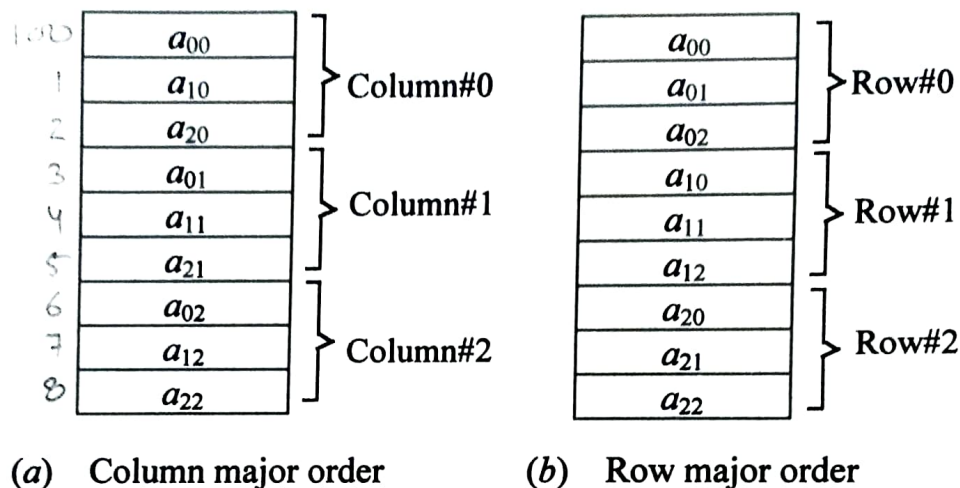
Let  $a$  be a two-dimensional  $m \times n$  array. Though  $a$  is pictured as a rectangular pattern with  $m$  rows and  $n$  columns, it is represented in memory by a block of  $m \times n$  sequential memory locations. However, the elements can be stored in two different ways —

- **Column major order** – the elements are stored column by column i.e.  $m$  elements of the first column are stored in first  $m$  locations, elements of the second column are stored in next  $m$  locations, and so on.
- **Row major order** – the elements are stored row by row i.e.  $n$  elements of the first row are stored in first  $n$  locations, elements of the second row are stored in next  $n$  locations, and so on.

Figure 3.7 shows these storage schemes for two-dimensional array  $A$  of size  $3 \times 3$ .

In Pascal, C/C++ the storage order is column major order, whereas in FORTRAN the storage order is row major order.





**Figure 3.7** Representation of two-dimensional array  $a$  of size  $3 \times 3$  in memory

Let us consider a two-dimensional array  $a$  of size  $m \times n$ . Further consider that the lower bound for the row index is  $lbr$  and for column index is  $lbc$ .

Like linear array, system keeps track of the address of first element only i.e. the *base address* of the array.

Using this base address, the computer computes the address of the element in the  $i$ th row and  $j$ th column i.e.  $loc(a[i][j])$ , using the following formulae:

### Column Major Order

$$loc(a[i][j]) = base(a) + w [m (j - lbc) + (i - lbr)] \quad \text{in general}$$

$$loc(a[i][j]) = base(a) + w (m \times j + i) \quad \text{in C/C++}$$

### Row Major Order

$$loc(a[i][j]) = base(a) + w [n (i - lbr) + (j - lbc)] \quad \text{in general}$$

$$loc(a[i][j]) = base(a) + w (n \times i + j) \quad \text{in C/C++}$$

where  $w$  is the number of bytes per storage location for one element of the array.