**UNIT-I**

 **ARRAYS**

**1. Abstract data types and the C++ class:**

 **1.1. An introduction to the C++ class:**

C++ provides an explicit mechanism, the CLASS, to support the distinction between specification and implementation and to hide the implementation of an ADT from its users.

The C++ class consists of four components:

1. A class name

2. Data members

3. Member functions

4. Levels of program access

**Example:**

**Program 1.1:**

**Class rectangle**

**{**

**Private:**

**int xlow,ylow,height,width**

**public:**

**rectangle ();**

**~rectangle ();**

**int getheight ();**

**int getwidth ();**

**};**

1. Class name: rectangle

2. Data members: xlow, ylow, height, width

3. Member functions: getheight () and getwidth ()

4. Levels of program access: public, private and protected

Data members are the data that makes up the class

Member functions are the set of operations that may be applied to the objects of a class

Levels of program access control the level of access to data members and member functions from program code that is outside the class

There are three levels

1. Public - Accessed anywhere from the class.
2. Private - Accessed from within its class or by a function or by a class that is declared to be a friend.
3. Protected – Access from within its class or from its subclass or by a friend class.

**1.2. Data Abstraction and Encapsulation in C++:**

**Data Abstraction** – data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

In C++ they provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally

For example, your program can make a call to the sort() without knowing what algorithm the function actually uses to sort the given values.

**Program 1.2:**

#include <iostream.h>

class adder

{

public:

 adder()

 {

 total=0;

 }

 void addnum(int number)

 {

 total += number;

 }

 int gettotal( )

 {

 return total;

 }

 Private:

 int total;

};

int main( )

{

 adder a();

a.addnum(10);

a.addnum(20);

a.addnum(30);

cout<< “total”<<a.gettotal()<<endl;

return o;

}

**Data encapsulation:**

Encapsulation is an object oriented programming concept that binds together the data and functions that manipulate the data, and keeps both safe from outside interference and misuse.

Data encapsulation led to the important concept of data hiding.

C++ supports the properties of encapsulation and data hiding through the creation of user defined data types, called classes.

**Program 1.3.**

Class box

{

 Public:

 Double getvolume()

 {

 return length\*breadth\*height;

 }

 Private:

 double length;

 double breadth;

 double height;

};

The variables length, breadth, height are private. This means that they can be accessed only by other members of the box class, and not by any other part of your program. This is one way encapsulation is achieved**.**

**1.3. Declaring class objects and invoking member functions:**

**Program 1.4:**

int rectangle::getheight()

{

 cout<<”enter height”;

 cin >>height

 return height;

}

int rectangle::getwidth()

{

 cout<<”enter width”;

 cin>>width;

 return width;

}

Void main( )

{

 Rectangle r;

 r.getheight();

 r.getwidth();

}

Program 1.4 shows a fragment of code that illustrates how class objects are declared and how member functions are invoked.

Class objects are declared and created in the same way that variables are declared or created.

Members of an object are accessed or invoked by using the component selection operators, a dot (.) for direct component selection and an arrow (🡪), for indirect component selection.

**1.4. Special class operations:**

**Constructors and Destructors:**

**Constructors:**

* Constructors and destructors are special member functions of a class.
* A constructor is a member function which initializes data members of an object. If a constructor is provided for a class, it is automatically executed when an object of that class is created.
* The advantage of constructor for a class is that all class objects are well defined as soon as they are created.
* A destructor is a member function which deletes data members immediately before object disappears.
* The constructor and destructor are declared in the program 1.1.
* The constructor must be declared as a public member function of its class.
* The name of the constructor is same as a class name.
* The constructor must not specify a return type or return a value.

**Program 1.5**

 rectangle::rectangle (int x, int y, int h, int w)

 {

 xlow=x;

 ylow=y;

 height=h;

 width=w;

}

Constructors may be used to initialize rectangle objects as follows

rectangle r(1,3,6,6);

rectangle \*S=new rectangle(2,4,3,1);

We can also have default constructor with no arguments

We have three types of constructors

1. Default constructor

2. Parameterized constructor

3. Copy constructor

**Destructors:**

* Destructors are automatically invoked when a class object goes out of scope or when a class object is deleted.
* The destructor must be declared as a public member function of its class.
* The name of the destructor is same as a class name prefixed with a ~.
* The destructor must not specify a return type or return a value.
* The destructor may not take any arguments.

**Operator overloading:**

C++ allows us to specify more than one definition for an operator that is called operator overloading.

 Operator overloading is used to perform mathematical operations between any two objects of the class.

Almost any operator can be overloaded in C++.except scope operator, sizeof operator, member selector, ternary operator.

Syntax:

returntype classname :: operator operatorsymbol(argument list)

{

 //function body

}

Operator is keyword

Operator overloading can be done by implementing a function which can be:

1. Member function
2. Non member function
3. Friend function

Restrictions for operator overloading

1. Precedence and associativity of an operator cannot be changed.
2. No new operators can be created only existing operators can be overloaded.
3. Cannot redefine the meaning of a procedure .you cannot change how integers are added.

**1.5. Miscellaneous topics:**

**STRUCT**

In C++, a struct is identical to a class, except that the default level of access is public; that is, if the struct definition of a data type does not specify whether a given member has public, private or protected access.

In a class, the default is private access. Thus, the C++ struct is a generalization of the C struct.

**UNION**

A union is a structure that reserves storage for the largest of its data members so that only one of its data members can be stored, at any time.

The struct or class structures reserve memory for all their data members. Thus, using a union results in a more memory-efficient program.

**STATIC**

A static class data member may be thought of as a global variable for its class.

One difference is that each class object does not have its own exclusive copy. There is only one copy of a static data member and all class objects must share it.

**1.6. ADTs and C++ classes:**

**ADT**:

An ADT is a programming language facility for organizing programs into modules using criteria that are based on the data structures of the program.

 The specification of the module should provide all information required for using the type, including the allowable values of the data and the effects the operations.

However, details about the implementation, such as data representations and algorithms for implementing the operations, are hidden within the module.

This separation of specification from implementation is a key idea of abstract data types

Example for ADT in C++ class

ADT for natural number

**Program 1.6**

Class naturalnumber( )

{

 Public:

 bool iszero( );

 naturalnumber add(naturalnumber y);

 bool equal(naturalnumber y);

 naturalnumber successor( );

 naturalnumber subtract(naturalnumber y);

};

**2. The array as an Abstract Data Type**

Many programmers view array as an consecutive set of memory locations.

An array is a set of pairs, <index,value>, such that each index that is defined has a value associated with it.

However when considering an ADT, we are more concerned with the operations that can be performed on array.

Two standard operations on array

1. Retrieves a value
2. Stores a value

**Abstract data type for general array**

**Program 1.7**

class generalarray

{

 public:

 generalarray(int j,rangelist list, float initvalue = defaultvalue;

 float retrieve(index i)

 void store(index i,float x);

};

In the above program we have three methods

1. Constructor generalarray is used to initialize the data members related to the class.It creates a new array of the appropriate size and type.
2. Retrieve accepts an index and returns the value associated with the index if the index is valid or an error if the index is invalid.
3. Store accepts an index, and a value of type float, and replaces the <index, oldvalue> pair with the <index, newvalue> pair.

**3. The Polynomial Abstract Data Type**

Arrays are not only data structures. We can also use them to implement other abstract data types. For instance let us consider the ordered, or linear, list.

The lists all contain items that are written in the form (a0, a1, a2......an-1)

We can perform many operations on lists, including:

1. Find the length, n, of the list.
2. Read the list from left to right(or right to left)
3. Retrieve the ith element, 0≤i<n.
4. Store a new value into the ith position, 0≤i<n.
5. Insert a new element at the position i, 0≤i<n, causing elements numbered i,i+1,i+2....,n-1 to become numbered i+1,i+2,.....n.
6. Delete the element at position i, 0≤i<n, causing elements numbered i+1,i+2,.....,n-1 to become numbered i,i+1,...n-2.

Let us jump right into a problem requiring ordered lists, which we shall solve by using one-dimensional arrays. The problem calls for building an ADT for the representation and manipulation of polynomials in a single variable

Two such polynomials are

a(x)=3x2+2x+4

b(x)=x8-10x5-3x3+1

The polynomial a(x) has 3 terms 3x2, 2x, and -4.

The coefficients of these terms are 3, 2, and -4.

The exponents of these terms are 2, 1 and 0.

A term of a polynomial may be represented as a (coefficient, exponent) pair.

For example (3, 2) represents 3x2.

A term whose coefficient is nonzero is called a nonzero term.

The degree of a polynomial is the largest exponent from among the nonzero terms.

Assume that we have two polynomials,

a(x)=∑aixi  and b(x)=∑bixi; then

 a(x)+b(x)=∑(ai+bi)xi

 a(x).b(x)=∑(aixi.∑(bjxj))

We begin with an ADT definition of a polynomial

**Abstract Data Type Polynomial**

**Program 1.8**

class polynomial

{

public:

polynomial( );

polynomial add(polynomial poly);

polynomial mul(polynomial poly);

}

**3.1 Polynomial Representation**

We are now ready to make some representation decisions.

A very reasonable decision would be to arrange the terms in decreasing order of exponent.

We have three types of representations

**Representation 1:**

Defining the private data members of polynomial as follows:

Private:

 int degree; //degree ≤ maxdegree

 float coef[maxdegree + 1]; //coefficient array

Where maxdegree is the constant that represents the largest- degree polynomial that is to be represented.

Now, if a is a polynomial class object and n≤maxdegree,then the polynomial a(x) above would be represented as:

a.degree = n;

a.coef[i] = an-i, 0≤i≤n

**Representation 2:**

Representation 1 requires us to know the maximum degree of the polynomials we except to work with and also quite wasteful in its use of computer memory.

Defining the private data members of polynomial as follows:

Private:

int degree;

float \*coef;

and adding the following constructor to polynomial:

polynomial :: polynomial(int d)

{

 degree = d;

 coef = new float[degree + 1];

}

**Representation 3:**

Although representation 2 solves the problems mentioned earlier, it does not yield a desirable representation.

In representation 2 to represent the term x1000+1 we need 1002 terms when we have only two nonzero terms

To overcome this problem, we store only nonzero terms. For this purpose, we define the class term as below.

class polynomial;

class term

{

 friend polynomial;

 private:

 float coef;

 int exp;

};

The private data members of polynomial are defined as follows:

private:

 term \*termarray;

int capacity;

int terms;

Representation 3 is definitely superior when there are many zero terms.

* 1. **Polynomial addition**

Function for adding two polynomials a and b, to obtain the sum c = a + b.

To store a and b values we use representation 3.

The basic loop of this algorithm consists of merging the terms of the two polynomials, depending upon the result of comparing exponents.

Then the loop for how the exponents are related with each other and perform the proper action. if polynomial runs out of terms, we must exit the loop

The remaining terms are directly copied to the result.

**Program 1.9**

//polynomial addition

#include<iostream>

using namespace std;

struct term

{

 int exp;

 float coef;

};

class polynomial

{

 int totalterms;

 term \*poly;

 public:

 polynomial(int);

 void read();

 void display();

 void Add(polynomial,polynomial);

 };

 polynomial::polynomial(int n)

 {

 poly=new term[n];

 }

 void polynomial::read()

 {

 cout<<"\nenter total number of terms:";

 cin>>totalterms;

 cout<<"\nenter terms:";

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

 {

 cout<<"\nEnter exponent:";

 cin>>poly[i].exp;

 cout<<"Enter coefficient:";

 cin>>poly[i].coef;

 }

 }

 void polynomial::display()

 {

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

 {

 if(i<totalterms-1)

 cout<<poly[i].coef<<"X^"<<poly[i].exp<<"+";

 else

 cout<<poly[i].coef<<"X^"<<poly[i].exp;

 }

 }

 void polynomial::Add(polynomial A,polynomial B)

 {

 int i=0,j=0,k=0;

 while((i<A.totalterms)&&(j<B.totalterms))

 {

 if(A.poly[i].exp==B.poly[j].exp)

 {

 poly[k].exp=A.poly[i].exp;

 poly[k].coef=A.poly[i].coef+B.poly[j].coef;

 i++;j++;k++;totalterms+=1;

 }

 else if(A.poly[i].exp>B.poly[j].exp)

 {

 poly[k].exp=A.poly[i].exp;

 poly[k].coef=A.poly[i].coef;

 i++;k++;totalterms+=1;

 }

 else

 {

 poly[k].exp=B.poly[j].exp;

 poly[k].coef=B.poly[j].coef;

 j++;k++;totalterms+=1;

 }

 }

 while(i<A.totalterms)

 {

 poly[k].exp=A.poly[i].exp;

 poly[k].coef=A.poly[i].coef;

 i++;k++;totalterms+=1;

 }

 while(j<B.totalterms)

 {

 poly[k].exp=B.poly[j].exp;

 poly[k].coef=B.poly[j].coef;

 j++;k++;totalterms+=1;

 }

 }

int main()

{

 polynomial A(10);

 polynomial B(10);

 polynomial C(10);

 A.read();

 cout<<"\npolynomial A:\n";

 A.display();

B.read();

 cout<<"\npolynomial B:\n";

 B.display();

 C.Add(A,B);

 cout<<"\nperforming addition...\n";

 cout<<"resultant polynomial:\n";

 C.display();

 return(0);

 }

Example polynomial expression

A(X)=3x4+6x3+2x+4

The above equation is stored in the poly array as given below

Poly

|  |  |  |  |
| --- | --- | --- | --- |
| **Poly[0]** | Poly[1] | Poly[2] | Poly[3] |
| **3** | **4** | 6 | 3 | 2 | 1 | 4 | 0 |

**Poly[0].Coef poly[0].exp**

**4. Sparse Matrices**

**4.1. Introduction**

A matrix is a mathematical object that arises in many physical problems

We are interested in studying ways to represent matrices so that the operations to be performed on them carried out efficiently.

A general matrix consists of m rows and n columns as shown in the below figure

Figure : Two matrices

 col 0 col 1 col 2

row 0 -27 3 4

row 1 6 82 -2

row 2 109 -64 11

row 3 12 8 9

 col 0 col 1 col 2 col 3 col 4

row 0 15 0 22 0 -15

row 1 0 11 3 0 0

row 2 0 0 0 -6 0

row 3 0 0 0 0 0

row 4 91 0 0 0 0

It is very natural to store a matrix in a two-dimensional array, say a[m][n]. Then we can work with any element by writing a[i][j], and this element can be found very quickly

Now we look at the second matrix in the above figure, we see that it has many zero entries. Such a matrix is called **sparse.**

In the above matrix only 7 entries out of 25 elements are nonzero, and that is sparse.

A sparse matrix requires us to consider an alternative form of representation.

Using an alternative representation, which stores only the nonzero elements, we can reduce both the space and time requirements considerably.

Before developing a particular representation for sparse matrices, we first must consider the operations that we want to perform on these matrices

The below program consists of our specification of the matrix ADT.

**Program 1.10**

class sparsematrix

{

 public:

 sparsematrix (int r, int c, int t);

 sparsematrix transpose( );

 sparsematrix add ( sparsematrix b);

};

**4.2. Sparse Matrix Representation**

Before implementing any of these operations**,** we mustestablish the representation of the sparse matrix.

Examine the below figure.

**Figure: Sparse matrix and its transpose stored as triples**

 **row col value**

**smarray[0] 0 0 15**

 **[1] 0 2 22**

 **[2] 0 4 -15**

 **[3] 1 1 11**

 **[4] 1 2 3**

 **[5] 2 3 -6**

 **[6] 4 0 91**

**Transpose of the above matrix**

 **row col value**

**smarray[0] 0 0 15**

 **[1] 0 4 91**

 **[2] 1 1 11**

 **[3] 2 0 22**

 **[4] 2 1 3**

 **[5] 3 2 -6**

 **[6] 4 0 -15**

We can uniquely identify the element in a matrix by a triple. i.e.,

< row, col, value>

**Row major representation:**

We require that these triples be stored by rows with the triples for first row first, followed by those of the second row, and so on.

We also require that all the triples for any row be stored so that the column indices are in ascending order.

**Column major representation:**

We require that these triples be stored by columns with the triples for first column first, followed by those of the second column, and so on.

We also require that all the triples for any column be stored so that the row indices are in ascending order.

In addition, to ensure that the operations terminate, we must know the number of rows and columns and the number of nonzero elements in the matrix.

We define this sparse matrix as given below

**Program 1.11**

class sparsematrix;

class matrixterm

{

 friend class sparsematrix;

 private:

 int row, col, value;

}

class sparsematrix

{

 private:

 int rows, cols, terms, capacity;

 matrixterm \*smarray;

}

**4.3. Transposing a Matrix**

**Simple Transpose**

* In the above section we have seen the transpose matrix for given matrix
* To transpose a matrix, we must interchange the rows and columns.
* This means if an element is at position [i][j] in the original matrix, then it is at position [j][i] in the transposed matrix. When i=j, the elements on the diagonal will remain unchanged.

For (each row i)

 Store (i, j, value) of the original matrix as (j, i, value) of the transpose;

* The difficulty is in not knowing where to put the element (j, i, value) until all other elements precede it have been processed

In the above we have seen the example of a transpose matrix

In that (0, 0, 15) which becomes (0, 0, 15)

 (0, 3, 22) which becomes (3, 0, 22)

 (0, 5, -15) which becomes (5, 0, -15)

 (1, 1, 11) which becomes (1, 1, 11)

* We can avoid this difficulty of not knowing where to place an element by changing the order in which we place elements into transpose

Consider the following strategy.

For (all elements in column j)

 Store ( i, j, value) of the original matrix as ( j, i, value)of the transpose;

* This says “find all elements in column 0 and store them in row 0, find all elements in column 1 and store them in row 1, etc.” Since the rows are originally in order, this means that we will loacate elements in the correct column order as well.

The function for transposing a matrix is given below

**Program 1.12**

sparsematrix sparsematrix :: transpose( )

{

 sparsematrix b ( cols, rows, terms);

 if (terms > 0)

 {

 int currentb=0;

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

 {

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

 {

 if (smarray[i].col==c)

 {

 b.smarray[currentB].row = c;

 b.smarray[currentB].col = smarray[i].row;

 b.smarray[current++].value = smarray[i].value;

 }

 }

 }

 }

 return b;

}

**Fast Transpose**

Let A be a sparse matrix of size *m\*n* with T non-zero elements. Its transpose will be

stored in matrix B. Let Freq and RowStartPos be two one-dimensional arrays of size *n*.

In Freq array, the frequency count of each column in matrix A is stored, and RowStart-

Pos will be computed and stored at the position where each row entry of matrix A is to b inserted in matrix B. Then, the RowStartPos is computed using Freq. The corresponding algorithm is given below



**4.4. Matrix Multiplication**

Given a and b,where a is m\*n and b is n\*p, the product matrix d has dimension m\*p. Its [i][j] element is

 **Dij=**$\sum\_{i=0}^{n-1}a\_{ik}b\_{kj}$

**Program 1.13**

**//Sparse Matrix Multiplication**

**#include<iostream.h>**

**using namespace std;**

**struct spM**

**{**

 **int row;**

 **int col;**

 **int value;**

**};**

**class sparse**

**{**

 **int Trows,Tcols,NonZero;**

 **spM \*Matrix;**

 **public:**

 **sparse(int,int,int);**

 **void read();**

 **void display();**

 **void transpose(Haritha);**

 **void product(Haritha,Haritha);**

**};**

**sparse::sparse(int r,int c,int terms)**

**{**

 **Trows=r;**

 **Tcols=c;**

 **NonZero=terms;**

 **Matrix=new spM[100];**

**}**

**void sparse::read()**

**{**

 **Matrix[0].row=Trows;**

 **Matrix[0].col=Tcols;**

 **Matrix[0].value=NonZero;**

 **for(int i=1;i<=NonZero;i++)**

 **{**

 **cout<<"Enter the row number:";**

 **cin>>Matrix[i].row;**

 **cout<<"Enter the column number:";**

 **cin>>Matrix[i].col;**

 **cout<<"Enter the value:";**

 **cin>>Matrix[i].value;**

 **}**

**}**

**void sparse::display()**

**{**

 **cout<<"Row\tCol\tval\n";**

 **for(int i=0;i<=NonZero;i++)**

 **{**

 **cout<<Matrix[i].row<<"\t"<<Matrix[i].col<<"\t"<<Matrix[i].value<<endl;**

 **}**

**}**

**void sparse::transpose(sparse B)**

**{**

 **int k=1;**

 **Matrix[0].row=B.Matrix[0].col;**

 **Matrix[0].col=B.Matrix[0].row;**

 **Matrix[0].value=B.Matrix[0].value;**

 **NonZero=B.NonZero;**

 **for(int i=0;i<B.Tcols;i++)**

 **{**

 **for(int j=1;j<=B.NonZero;j++)**

 **{**

 **if(B.Matrix[j].col==i)**

 **{**

 **Matrix[k].row=B.Matrix[j].col;**

 **Matrix[k].col=B.Matrix[j].row;**

 **Matrix[k].value=B.Matrix[j].value;**

 **k++;**

 **}**

 **}**

 **}**

**}**

**void sparse::product(sparse A,sparse C)**

**{**

**sparse B(0,0,0);**

**Matrix[0].col=A.Tcols;**

**Matrix[0].row=A.Trows;**

**int RA=A.Matrix[1].row;int CB;**

**int RI=1;**

**int RBe=1;int CI;int k=1;int sum=0;**

**B.transpose(C);**

**while(RI<=A.NonZero)**

**{**

 **CI=1;**

 **CB=B.Matrix[1].row;**

 **while(CI<=B.NonZero)**

 **{**

 **if(A.Matrix[RI].row!=RA)**

 **{ if(sum!=0)**

 **{**

 **Matrix[k].row=RA;**

 **Matrix[k].col=CB;**

 **Matrix[k].value=sum;**

 **k++;**

 **}**

 **sum=0;**

 **RI=RBe;**

 **while(B.Matrix[CI].row==CB)**

 **CI++;**

 **CB=B.Matrix[CI].row;**

 **}**

 **else if(B.Matrix[CI].row!=CB)**

 **{ if(sum!=0)**

 **{**

 **Matrix[k].row=RA;**

 **Matrix[k].col=CB;**

 **Matrix[k].value=sum;**

 **k++;**

 **}**

 **sum=0;**

 **RI=RBe;**

 **CB=B.Matrix[CI].row;**

 **}**

 **else**

 **{**

 **if(A.Matrix[RI].col<B.Matrix[CI].col)**

 **RI++;**

 **else if(A.Matrix[RI].col==B.Matrix[CI].col)**

 **{**

 **sum+=A.Matrix[RI].value\*B.Matrix[CI].value;**

 **RI++;CI++;**

 **}**

 **else**

 **CI++;**

 **}**

 **}**

 **if(sum!=0)**

 **{**

 **Matrix[k].row=RA;**

 **Matrix[k].col=CB;**

 **Matrix[k].value=sum;**

 **sum=0;**

 **k++;**

 **}**

 **while(A.Matrix[RI].row==RA)**

 **RI++;**

 **RBe=RI;**

 **RA=A.Matrix[RI].row;**

 **}**

 **Matrix[0].value=k-1;**

 **NonZero=k-1;**

**}**

**int main()**

**{**

 **sparse A(3,3,4);**

 **sparse B(3,3,4);**

 **sparse D(0,0,0);**

 **A.read();**

 **cout<<"\nMatrix A\n";**

 **A.display();**

 **B.read();**

 **cout<<"\nMatrix B\n";**

 **B.display();**

 **cout<<"\nPerforming Multipliction.......\n";**

 **cout<<"Resultant Matrix is:\n";**

 **D.product(A,B);**

 **D.display();**

 **return(0);**

**}**

**5. Representation of arrays**

* Multidimensional arrays are usually implemented by storing the elements in a one dimensional array.
* In this section, we develop a representation in which an arbitrary array element, say a[i1][i2][i3].....[in],gets mapped onto a position in a one-dimensional C++ array so that it can be retrieved efficiently.
* In addition to being able to retrieve array elements easily, it is also necessary to be able to determine the amount of memory to be reserved for a particular array.
* The mechanism for allocating memory is much dependant on a language. Regardless of machine and language dependency, when the space is actually allocated, the location of an entire block of memory is referenced by the base address of the first location. The remaining elements are stored sequentially at a fixed distance apart, say, by a constant C.
* Here, C depends on the size of the element, that is, the number of locations required per element, and also on the addressing of these locations.
* The address of the ith element is calculated by the following formula:

 (Base address) + (Offset of ith element from base address)

Offset of ith element = (Number of elements before ith element)

 \* (Size of each element)

 Address of A[i] = Base + i \*Size of element

In brief, the Array\_A[N] is implemented as follows:

1. Array\_A is the name of the object/structure and is associated with a base (starting) address in memory.

2. The [N] notation specifies the number of array elements from the beginning (offset), which starts at zero.

3. The address of the ith element is then computed as base + i \*(Size of element), where Size of element depends on the data type.



Row major representation



Column major representation

 

**Address calculation for one dimensional array**

****

**Address calculation for two dimensional array**

****

****

**Address calculation for three dimensional array**



Repeating in this way, the address for a[i1][i2],.....[in] is

 X+i1u2u3.....un

+i2u3u4.....un

+i3u4u5.....un+........+in-1un+in

 $=x+\sum\_{j=1}^{n}i\_{j}a\_{j}$