UNIT – I

Arrays: Abstract Data Types and the C++ Class, An Introduction to C++ Class- Data Abstraction and Encapsulation in C++- Declaring Class Objects and Invoking Member Functions- Special Class Operations- Miscellaneous Topics- ADTs and C++Classes, The Array as an Abstract Data Type, The Polynomial Abstract Data type- Polynomial RepresentationPolynomial Addition. Spares Matrices.

.

***

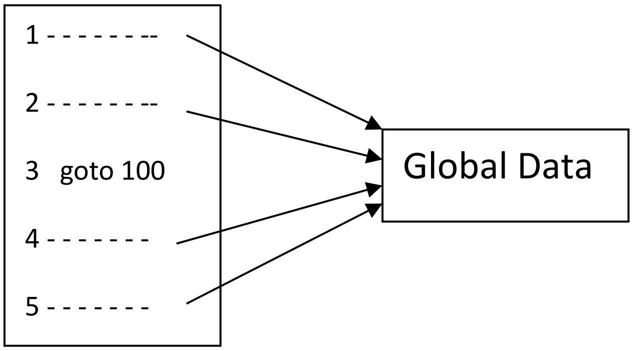
PROGAMMING PARADIGMS

Programming paradigms are categorized into different types such as:

- ✓ Monolithic programming
- ✓ Procedure-oriented programming
- ✓ Structured programming
- ✓ Object-oriented programming

Monolithic Programming

In this programming approach, entire programming instructions are designed in a single file and are in sequential order.



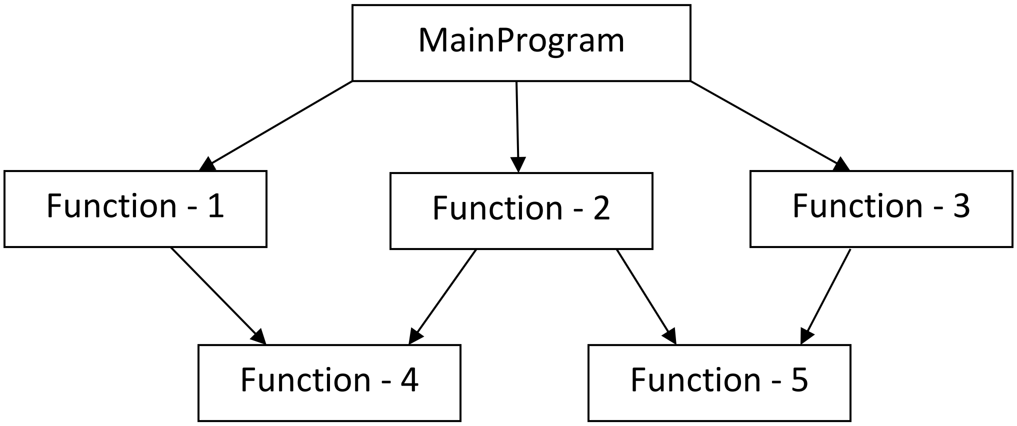
- ➢ Programs are organized in sequential order and all data items are global.
- ➢ Program controls are through goto statements.
- ➢ Suitable for small and simple applications.

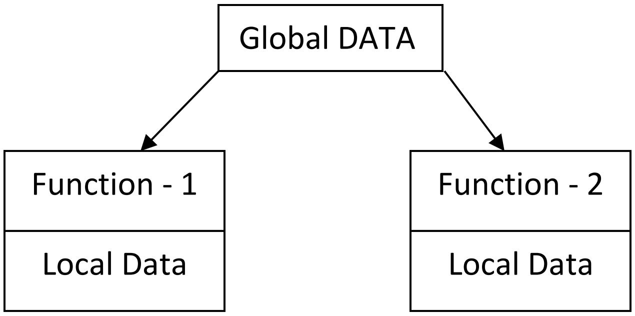
<u>Examples:</u> Machine language, Assembly language.

<u>Drawbacks:</u> 1. Difficult to maintain program code
2. Data security problem

Procedure-oriented Programming

In procedure-oriented programming approach, the given problem statement is divided into small tasks known as functions. Each procedure is individually solved and finally those are integrated. The typical structure of a procedure-oriented programming is as follows:





- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Employs top-down approach in program design.

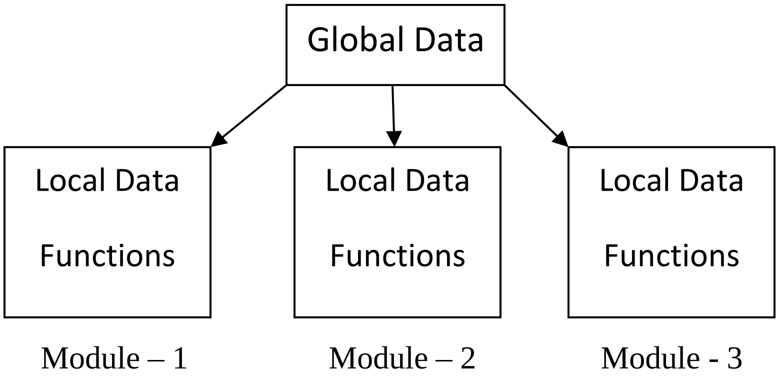
Examples: COBOL, FORTRAN etc,

Drawbacks: 1. Difficult to maintain program code

2. Data security problem

Structured Programming

In structured programming approach, the given problem statement is divided into small modules and each module has a set of functions of related types.



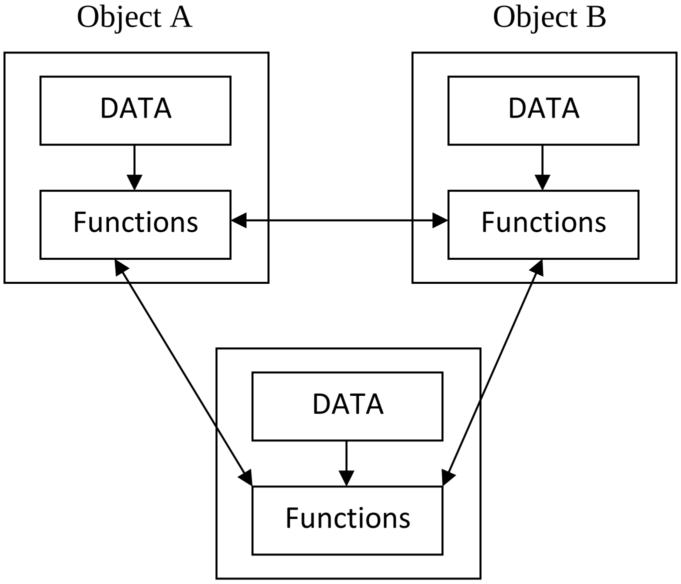
- Application is divided into individual modules.
- Each modules consists a set of functions and has processing logic.
- User defined data types and functions are introduced.

Examples: PASCAL, C etc,

Drawbacks: 1. It does not support real world applications very well
2. Data security problem

Object-oriented Programming

In object-oriented programming, problem is decomposed into a number of entities called **objects** and then builds data and functions around these objects. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system.



Object C

- Emphasis on data rather than procedure.
- Programs are divided into objects.
- Data structures are designed such that they characterize the objects.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design

Examples: C++, JAVA, SMALL TALK etc,

BASIC CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

Basic concepts of object-oriented programming approach are:

- ✓ Objects
- ✓ Classes
- ✓ Data Abstraction and Encapsulation
- ✓ Inheritance
- ✓ Polymorphism
- ✓ Dynamic binding
- ✓ Message Passing

C++ Introduction

C++ is an object-oriented programming language developed by Bjarne Stroustrup at AT & T Bell Laboratories, USA, in the early 1980's. Initially the language was named as **'C with classes'**. Later in 1983, the name was changed to **C++.** The idea of C++ comes from the C increment operator ++ hence C++ is an incremented version of C.

Comments

In C++ two types of comments are available such as: Single line comment and Multiline comments.

Single line comments: Comments start with a double slash symbol (//) and terminate at the end of the line.

Example:

```
// C++ Example Program
// Designed as Sample Program
```

Multiline comments: Comments start with /* and ends with */ are treated as multiline comments.

Example:

```
/* C++ Example Program
   Designed as Sample Program */
```

iostream.h File

iostream.h header file supports various predefined objects that supports input and output statements.

Example: #include<iostream.h>

main() Function

main() is a special function used by the C++ system to tell the compiler where the actual program execution starts and terminates. Every C++ program must have exactly one main() function.

In C++, main() returns an integer value to the operating system. Therefore, every main() in C++ should end with a return 0 statement.

Extraction (or) Get From Operator

The operator **>>** is known as **extraction** or **get from** operator. Extraction operator is used with a predefined input object known as **cin** that corresponds to the standard input stream Keyboard.

Example:

```
int K;
cin>>K;
```

Here, the operator extracts the value from the keyboard and assigns it to the variable on its right.

Insertion (or) Put to Operator

The operator << is known as **insertion** or **put to** operator. Insertion operator is used with a predefined output object known as **cout** that corresponds to the standard output stream Monitor.

Example:

```
int K=10;
cout<<K;
```

Here, the operator sends the contents of the variable on its right to the object on its left.

Structure of a C++ program

A C++ program would contain four sections such as:

Include Header Files
Class Declaration
Member functions Definitions
Main function Program

Save	:	F2	:	Save the C++ program file with **.cpp** extension
Compilation	:	Alt F9 / F9		
Execution	:	Ctrl F9		
Result View	:	Alt F5		

```
// ADDITION OF GIVEN TWO VALUES

#include<iostream.h>
#include<conio.h>

int main()
{
        int x,y,z;
        clrscr();
        cout<<"Enter Two Values = ";
        cin>>x>>y;
        z=x+y;
        cout<<"Addition Result = "<<z;
        return 0;
}
```

CLASS

A class is a new abstract data type that shows a way to bind the data and the associated functions together. Class specification has two parts:

1. Class declaration

2. Class function definitions

The **class declaration** describes the type and scope of its members. The **class function definitions** describe how the class functions are implemented.

The general form of a class declaration is:

Syntax:

```
class Class_Name
{
        private: variable declarations;
                 function declarations;

        public:  variable declarations;
                 function declarations;
};
```

- Every class declaration starts with the keyword class followed by name of the class surrounded by a pair of braces and terminated with a semicolon.
- The class body contains variable declarations and function declarations collectively known as class members.
- The variables declared inside the class are known as **data members** and the functions are known as **member functions**.
- The class members have been declared with the access specifiers **private, public** and **protected**. The class members that have been declared as private can be accessible through

the public members of the same class. In C++, if the access control is not defined, by default all the members are private.

Example:

```
class Product
{
        private: int Number;
                     float Cost;
        public: void getdata();
                   void putdata();
};
```

OBJECT

Once a class has been created, we can create variables of that type by using the class name as:

Syntax: **ClassName VariableName / ObjectName;**

Example: **Product obj;**

In C++, the class variables are known as **objects.** In general, object is an instance of a class. A class has logical appearance, whereas an object has physical existence.

Accessing Class Members

The private data of a class can be accessed only through public member functions of that same class. A variable declares as public can be accessed by the objects directly. In general, class members can be accessed with the class object as:

Syntax: **ObjectName . ClassMembers**

Example: **obj.getdata( );**

DEFINING MEMBER FUNCTIONS

Member functions can be defined in two places:

- Outside the class definition
- Inside the class definition

Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. They should have a function header and a function body. The general form of a member function definition is:

Syntax:

```
ReturnType ClassName :: FunctionName(Arguments Declaration)
{
        Function Body
}
```

Here,

- The symbol :: is called the **scope resolution** operator.
- ClassName :: tells the compiler that the function FunctionName belongs to the class ClassName.

Example:

```
void Product::getdata()
{
        cout<<endl<<"Enter Product Number = ";
        cin>>Number;
        cout<<endl<<"Enter Product Cost = ";
        cin>>Cost;
}
```

// Example C++ program to read and print product details with class

```
#include<iostream.h>
#include<conio.h>

class Product
{
        private: int Number;
                float Cost;
        public: void getdata();
                void putdata();
};

void Product::getdata()
{
        cout<<endl<<"Enter Product Number = ";
        cin>>Number;
        cout<<endl<<"Enter Product Cost = ";
        cin>>Cost;
}
```

```
void Product::putdata()
{
        cout<<endl<<"PRODUCT ID      ="<<Number;
        cout<<endl<<"PRODUCT PRICE ="<<Cost;
}

int main()
{
        Product obj;
        clrscr();
        obj.getdata();
        obj.putdata();
        return 0;
}
```

<u>Inside the Class Definition</u>

Member function definitions can also be defined inside the class declaration. When a function is defined inside the class, it is treated as **inline** function.

// Example C++ program to read and print product details with class

```
#include<iostream.h>
#include<conio.h>

class Product
{
        private: int Number;
                float Cost;
        public: void getdata()
                {
                        cout<<endl<<"Enter Product Number = ";
                        cin>>Number;
                        cout<<endl<<"Enter Product Cost = ";
                        cin>>Cost;
                }
                void putdata()
                {
                        cout<<endl<<"PRODUCT ID    ="<<Number;
                        cout<<endl<<"PRODUCT PRICE ="<<Cost;
                }
};
```

```
int main()
{
        Product obj;
        clrscr();
        obj.getdata();
        obj.putdata();
        return 0;
}
```

CONSTRUCTORS

A constructor is a special member function used to initialize the objects of the class at the time of their creation. Constructor has the same name as class name. The constructor is invoked automatically whenever an object of the class is created.

Example:

```
class Demo
{
        private: int X;
                 float Y;

        public:  Demo( )              // Constructor
                 {
                         X = 0;
                         Y = 0.0;
                 }
                 void printdata();
};
```

The constructor functions have some special characteristics. These are:

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return values even void.
- Like other C++ functions, they can have default arguments.
- Constructors can be overloaded.

TYPES OF CONSTRUCTORS

In C++ language, three types of constructors are available such as:

A) Default constructors
B) Parameterized constructors
C) Copy constructors

A) Default Constructors

The constructor without any arguments is called a default constructor. If no such constructor is defined, then the compiler automatically supplies a default constructor.

Example:

```
#include<iostream.h>
#include<conio.h>
class Demo
{
        private: int X;
                 float Y;
        public:  Demo( )                // Default Constructor
                 {
                        X = 0;
                        Y = 0.0;
                }
                void putdata()
                {
                        cout<<endl<<"X Value ="<<X;
                        cout<<endl<<"Y Value ="<<Y;
                }
};

int main()
{
        Demo obj;
        clrscr();
        obj.putdata();
        return 0;
}
```

B) Parameterized Constructors

Constructors those are created with arguments are called parameterized constructors. At the time of creating an object for the class, values must be passed to the constructors.

Example:

```
#include<iostream.h>
#include<conio.h>
class Demo
{
        private: int X;
                float Y;
        public: Demo(int A,float B)   // Parameterized constructor
                {
                        X=A;
                        Y=B;
```

```
                    }
                    void putdata()
                    {
                            cout<<endl<<"X Value ="<<X;
                            cout<<endl<<"Y Value ="<<Y;
                    }
        };

        int main()
        {
                Demo obj(12,567.97);
                clrscr();
                obj.putdata();
                return 0;
        }
```

C) Copy Constructors

When the reference of an object is passed as an argument to the constructors, such types of constructors are called copy constructors.

Example:

```
#include<iostream.h>
#include<conio.h>
class Demo
{
        private: int X;
        public: Demo(int A)
        {
                X=A;
        }

        Demo(Demo &K)     // Copy constructor
        {
                X=K.X;
        }

        void display()
        {
                cout<<endl<<"X Value ="<<X;
        }
};

int main()
{
        Demo obj1(10);
        Demo obj2(obj1);
        clrscr();
        obj1.display();
        obj2.display();
        return 0;
}
```

DESTRUCTORS

Destructor is also a member function used to destroy the objects created by the constructors. The destructor name is also the same as the class name but it is preceded by a tilde (~) symbol.

Example:

```
~Demo( )
{
        cout<<"Object Destroyed";
}
```

A destructor never takes any argument nor does it return any value. It will invoked implicitly by the compiler whenever the program execution completed.

Some of the important characteristics of a destructor are:

- Destructors do not require any arguments and does not return any value.
- Destructors are implicitly invoked by the compiler whenever the program execution terminated.
- Destructors cannot be overloaded.

Example:

```
#include<iostream.h>
#include<conio.h>

class Demo
{
        public: Demo()
                {
                        cout<<endl<<"Object Created";
                }
                ~Demo()
                {
                        cout<<endl<<"Object Destroyed";
                }
};

int main()
```

```
{
        Demo K;
        return 0;
}
```

DATA ENCAPSULATION

The wrapping of data and functions into a single unit is known as encapsulation. With encapsulation, the data is not accessible to the outside world, and only those functions, which are wrapped in the class, can access it. These functions provide the interface between the object's data and the program. Some time, it is also referred as data hiding or information hiding.

DATA ABSTRACTION

Data abstraction is a mechanism to present only the essential features without including their background details or explanation.

Classes use the concept of abstraction with data members and member functions to operate on those data members.

DATA STRUCTURES

The logical or mathematical model of a particular organization of data is called as a data structure.

(OR)

The class of data that can be characterized by its organization is known as a data structure.

Data structures are classified into two categories such as:

a) Linear data structures
b) Non-linear data structures

a) Linear data structures: A data structure is said to be linear if the values are arranged in a sequence order i.e., in linear fashion. Here, there is a relationship between the adjacent elements of the list.

Examples: Arrays, Stack, Queues, Linked list etc.

b) Non-linear data structures: A data structure is said to be non-linear if the values are arranged without any sequence order i.e., in non-linear fashion. Here, there is no relationship between the adjacent elements of the list.

Examples: Trees, Graphs, Sets, Tables etc.

APPLICATIONS OF DATA STRUCTURES:

Data structures are extensively used in different application areas such as:

- Compiler design
- Operating system
- Numerical analysis
- Graphics
- Artificial intelligence
- Simulation etc.

ABSTRACT DATA TYPE

Abstract Data Type (ADT) is a specification for the type of values that a data type can store and the operations that can be performed on those values. It is just a specification without any details on the implementation part.

In C++, classes use the concept of data abstraction, they are known as Abstract Data Types.

Examples: Array ADT, Stack ADT, Queue ADT etc.

ARRAYS

An array is a collection of homogeneous/similar data type elements those are stored in successive memory locations. An array is referred by specifying the array name followed by one or more subscripts, with each subscript is enclosed in square brackets. The number of subscripts determine the dimensionality of the array. Depending on the number of subscripts used, arrays can be classified into different types as:

1. Single (or) One dimensional arrays
2. Double (or) Two dimensional arrays
3. Multi dimensional arrays

1. Single (or) One dimensional arrays

Let ‘m’ is the size of an array, the one dimensional array can be defined as – “One dimensional array is a collection of m homogeneous data elements those are stored in m successive memory locations”.

The general format of a single dimensional array is:

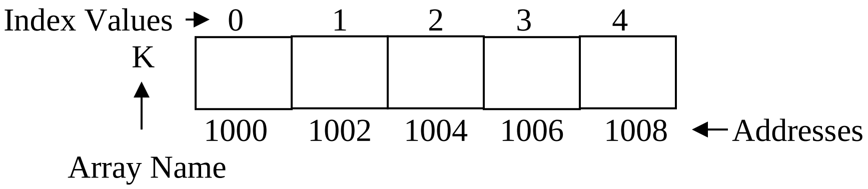
Syntax: **datatype ArrayName[size];**

Where,

- datatype specifies the type of the elements that will be stored in the array.
- ArrayName specifies the name of the array that follows same rules as a valid identifier.
- size indicates the maximum number of elements that can be stored inside the array.

Example: **int K[5];**

For this, the memory allocation will be:



Note:

1. The elements in an array is stored in successive memory locations.
2. The elements of array are referred respectively by an index.
3. Elements of the array are denoted as K[0], K[1], - - - - - K[N-1].

2. Double (or) Two dimensional arrays

Let ‘m’ is the row size and ‘n’ is the column size, then a double dimensional array can be defined as – “Double dimensional array is a collection of m x n homogeneous data elements those are stored in m x n successive memory locations”.

The general format of a double dimensional array is:

Syntax: **datatype ArrayName[size1][size2];**

Where,

- datatype specifies the type of the elements that will be stored in the array.
- ArrayName specifies the name of the array that follows same rules as a valid identifier.

- size1specifies row size i.e., number of rows.
- size2specifies column size i.e., number of columns.

Example: **int k[3][4];**

For this, memory allocation will be:

		0	1	2	3
	0				
K	1				
	2				

For double dimensional arrays, memory is allocated in terms of table format that contains collection of rows and columns. So, that double dimensional arrays are useful for matrix representation of given elements.

3. Multidimensional arrays

Multidimensional array uses three or more dimensions. Let m1, m2, - - - , mn are the sizes, then a multidimensional array can be defined as – "Multidimensional array is a collection of m1 x m2 x - - - - x mn homogeneous data elements those are stored in m1xm2x- - - - x mn successive memory locations".

The general format of a multidimensional array is:

Syntax: **datatype ArrayName[size1][size2] - - - - - - [sizen];**

Example: **int k[2][3][4];**

ARRAY AS AN ABSTRACT DATA TYPE

An array is a fundamental abstract data type. Specification for an array ADT can be shown as:

ADT Array
{
Instances: Collection of homogenious data elements

Operations:

```
        Create(): Creates an array with the specified number of elements.
        Insertion(Key,Pos): Inserts an element Key at the specified position Pos of the array.
        Deletion(Pos): Delete the specified position Pos element of the array.
        Get(Index): Return the existent element at Index position of the array.
        Set(Index,Key): Replaces the Value with Key at the specified Index position of the array.
        Display(): Displays all elements of the array.
}
```

Implementations

Let K is a single dimensional array with a maximum size as 50 elements.

```
class ArrayADT
{
        int K[50],N;
        public: void Create();
                void Insertion(int,int);
                int Deletion(int);
                void Set(int,int);
                int Get(int);
                void Display();
};
```

Create(): This procedure creates a single dimensional array with the specified number of elements. For this, implementation code can be shown as:

```
void ArrayADT::create()
{
        int i;
        cout<<endl<<"Enter How Many Elements = ";
        cin>>N;
        cout<<"Enter "<<N<<" Elements = ";
        for(i=0;i<N;i++)
        cin>>K[i];
}
```

Insertion(Key , Pos): This procedure inserts a new element Key at the specified position Pos of the array K. For this, implementation code can be shown as:

```
void ArrayADT :: Insertion(int Key , int Pos)
{
        for(int i=N-1;i>=Pos-1;i--)
        K[i+1] = K[i];
        K[Pos-1] =  Key;
```

```
          N = N+1;
}
```

Deletion(Pos): Function removes the specified position element from the array K. For this, implementation code can be shown as:

```
int ArrayADT :: Deletion(int Pos)
{
          int i,P;
          P = K[Pos-1];
          for(i=Pos-1;i<=N-2;i++)
          K[i] = K[i+1];
          N = N-1;
          return P;
}
```

Set(Index , Key): This procedure replaces a specified Index element with new element Key at the array K. For this, implementation code can be shown as:

```
void ArrayADT :: Set(int Index,int Key)
{
          K[Index] = Key;
}
```

Get(Index): Funtion returns a specified Index element of the array K. For this, implementation code can be shown as:

```
int ArrayADT :: Get(int Index)
{
          return K[Index];
}
```

Display(): This procedure prints all the elements of the array K. For this, implementation code can be shown as:

```
void ArrayADT::Display()
{
          for(int i=0;i<=N-1;i++)
          cout<<"        "<<K[i];
}
```

POLYNOMIAL REPRESENTATIONS

The general form of a polynomial equation with a simple variable is:

$$\mathbf{P(x) = a_{m-1}\, x^{em-1} + a_{m-2}\, x^{em-2} + - - - - - - -- + a_0\, x^{e0}}$$

Where,

a_i are nonzero coefficients and the e_i are nonnegative integer exponents.

The largest exponent of a polynomial is called it degree.

Example: P(x) = $3x^{14} – 7x^8 + 1$

A polymial equation can be represented using arrays and linked lists.

Polynomial as Array Representation

Assume the exponent of the expression is arranged from index 0 to Degree, which is represented by the index values, and then respected coefficient values are stored in those particular index positions of the array. For this, two types of formats are possible.

Format-I: **Single dimensional Array:** In this format index values of the array is used as exponent value indication.

Example: Given $P(x) = 18x^7 – 41x^6 + 75x^4 – 5x + 3$

Here, it can be represented as:

	0	1	2	3	4	5	6	7
P =	3	-5	0	0	75	0	-41	18

Format-I I: **Double dimensional Array:** In this format exponent values are stored in one row and the corresponding coefficient values are stored in another row.

Example: Given $P(x) = 18x^7 – 41x^6 + 75x^4 – 5x + 3$

Here, it can be represented as:

	0	1	2	3	4
P =	7	6	4	1	0
	18	-41	75	-5	3

POLYNOMIAL AS AN ABSTRACT DATA TYPE

Specification for a polynomial ADT can be shown as:

```
ADT Polynomial
{
	Instances:	Collection of coefficient values

	Operations:

	ReadPoly(Degree): Read coefficient values from the user and creates polynomial.
	SetCoeff(Coef,Exp): Sets a coefficient value at the specified exponent position.
	GetCoeff(Exp): Return coefficient value at the given exponent positon.
	Add(Polynomial A,Polynomial B): Add two polynomials A and B.
	Sub(Polynomial A,Polynomial B): Subtracts two polynomials A and B.
	Mul(Polynomial A,Polynomial B): Multiply two polynomials A and B.
	Display(): Displays Polynomials without zero coefficients.
}
```

<u>Implementations</u>

Let K is a single dimensional array with a maximum size as 50 elements.

```
class PolynomialADT
{
	int K[50],Degree;
	public: void ReadPoly(int);
		void Setcoeff(int,int);
		int Getcoeff(int);
		void Add(PolynomialADT,PolynomialADT);
		void Display();
};
```

<u>ReadPoly(Degree):</u> This procedure accepts coefficient values from the user and creates polynomial based on its degree.

```
void PolynomialADT::ReadPoly(int N)
{
	int i;
	Degree=N;
	for(i=0;i<=Degree;i++)
	{
	   cout<<"Enter Coefficient For Exponent "<<i<<" = ";
	   cin>>K[i];
	}
}
```

SetCoeff(Coef,Exp): This procedure sets a coefficient value at the specified exponent position.

```
void PolynomialADT::Setcoeff(int Coef,int Exp)
{
        K[Exp]=Coef;
}
```

GetCoeff(Exp): Funtion returns coefficient value of the given exponent position.

```
int PolynomialADT::Getcoeff(int Exp)
{
        return K[Exp];
}
```

Display(): This procedure displays Polynomials without zero coefficients.

```
void PolynomialADT::Display()
{
        int i;
        for(i=Degree;i>=0;i--)
        {
           if(K[i]!=0)
           {
             cout<<K[i]<<" X ^ "<<i;
             if(i!=0)
             cout<<" + ";
           }
        }
}
```

Add(Polynomial A,Polynomial B): This procedure adds two polynomials A and B.

```
void PolynomialADT::Add(PolynomialADT A,PolynomialADT B)
{
          Degree=A.Degree;
          for(int i=0;i<=Degree;i++)
          K[i]=A.K[i]+B.K[i];
          Display();
}
```

SPARSE MATRICES

A sparse matrix is a matrix representation in which number of zero elements is greater than the number of non-zero elements.

Example:

$$\begin{matrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{matrix}$$

Special kinds of sparse matrices are Diagonal matrices, Lower triangular matrices, Upper triangular matrices etc.

Lower triangular matrix is a special kind of square matrix in which all the entries above the main diagonal are zeros.

Example:

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 7 & 2 & 0 & 0 \\ 4 & 6 & 3 & 0 \\ 2 & 7 & 9 & 4 \end{matrix}$$

Upper triangular matrix is a special kind of square matrix in which all the entries below the main diagonal are zeros.

Example:

$$\begin{matrix} 1 & 5 & 8 & 4 \\ 0 & 2 & 8 & 6 \\ 0 & 0 & 3 & 5 \\ 0 & 0 & 0 & 4 \end{matrix}$$

Diagonal matrix is a special kind of square matrix in which every element except the principle diagonal elements is zero.

Example:

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

<u>Note:</u> Sparse matrix can be represented using Triplet and Linked list.

<u>Triplet Representation of a Sparse Matrix</u>

In this representation, consider only non-zero values along with their row and column index values. Here each non-zero element is represented with a triplet form as: **<Row, Column, Value>**

In triplet representation, consider a table that consists three columns for storing Row, Column and Element values and number of rows based on the non-zero entries. In first row, we must specify the number of rows, columns and non-zero entries of the given sparse matrix.

<u>Example:</u> Consider the sparse matrix as:

$$\begin{matrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{matrix}$$

For this, in triplet format it can be represented as:

Rows	Columns	Elements
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

END