UNIT – I

1. ABSTRACT DATA TYPE (ADT)

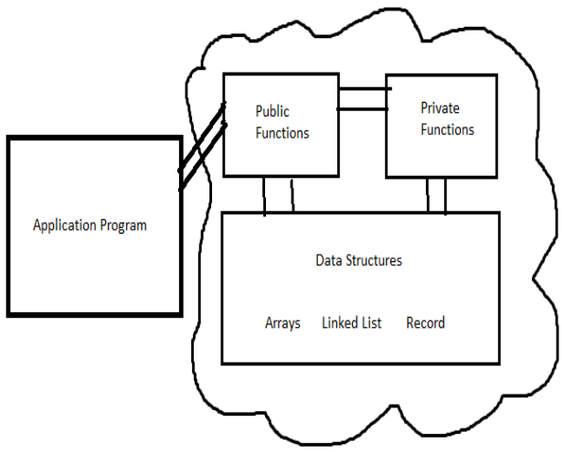
An ADT refers to a set of data values and associated operations that are specified accurately. ADT consists of set of definitions that allow us to use the function while hiding the implementation (i.e.; ADT is independent of implementation)

ADT model: The representation of the ADT model is shown below

ADT consists of two different parts:

1) Data Structures
2) Functions

- Data Structures and Functions don't come within the scope of Application Program.
- Data Structures are available to all of the ADT functions i.e. Data Structures and Functions are within the scope of each other.
- The programming interface can only access public functions.
- For each ADT operation, there is an algorithm that performs specific task.
- Only operation name and parameters are available to application program but implementation is hidden.



ADT Model

AN INTRODUCTION TO 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. However, ti is the programmer's responsibility to use the class mechanism judiciously so that it does, in fact, represent an ADT. The C++ class consists of four components:

1) a class name: (e.g. Rectangle)

2) Data members: the data that makes up the class (e.g. x1, y1, h and w).

3) Member functions: the set of operations that may be applied to the objects of a class (e.g. GetHeight(), Getwidth()).

4) Levels of program access: these control the level of access to data members and member functions from program code that is outside the class. There are three levels of access to class members: public, protected and private.

Any public data member (member function) can be accessed (invoked) from anywhere in the program. A private data member (member function) can only be accessed (invoked) from within its class or by a function or a class that is declared to be a friend. A protected data member (member function) can only be accessed (invoked) from within its class or from its subclasses or by a friend.

```
// In the header file Rectangle.h
class Rectangle {
public:        // the following members are public
   // The next four members are member functions
   Rectangle();      // constructor
   ~Rectangle();     // destructor
   int GetHeight();  // returns the height of the rectangle
   int GetWidth();   // returns the width of the rectangle
private:          // the following members are private
   // the following members are data members
   int x1, y1, h, w;
   // (x1, y1) are the coordinates of the bottom left corner of the rectangle
   // w is the width of the rectangle; h is the height of the rectangle
};
```

Program 1: Definition of the C++ class Rectangle

DATA ABSTRACTION AND ENCAPSULATION IN C++

Data encapsulation is enforced in C++ by declaring all data members of a class to be **private** (or **protected**). External access to data members, if required, can be achieved by defining member functions that get and set data members. In Program 2.1, *GetHeight*() and *GetWidth*() are used to access the private data members *h* and *w*. Member functions that will be invoked externally are declared **public**; all others are declared **private** or **protected**.

Next, we discuss how the specification of the operations of a class is separated from their implementation in C++. The specification, which must be contained inside the public portion of the class definition of the ADT, consists of the names of every public member function, the type of its arguments, and the type of its result (This information about a function is known as its *function prototype*). There should also be a description of what the function does, which does not refer to the internal representation or implementation details. This requirement is quite important because it implies that an abstract data type is *implementation-independent*. This description may be achieved in C++ by using comments to describe what each member function does (like the VCR instruction manual mentioned in Chapter 1). Finally, the specification of an operation is physically separated from its implementation by placing it in an appropriately named header file (e.g. the contents of Program 2.1 are placed in *Rectangle.h*). The implementations of the functions are typically placed in a source file of the same name (e.g. the contents of Program 2.2 are placed in *Rectangle.C*). Note that C++ syntax does allow you to include the implementation of a member function inside its class definition. In this case, the function is treated as an *inline* function.

```
// In the source file Rectangle.C
#include "Rectangle.h"

// The prefix "Rectangle::" identifies GetHeight() and GetWidth() as member functions
// belonging to class Rectangle. It is required because the member functions
// are implemented outside their class definition

int Rectangle::GetHeight() { return h;}
int Rectangle::GetWidth() { return w;}
```

Program 2.2: Implementation of operations on *Rectangle*

DECLARING CLASS OBJECTS AND INVOKING MEMBER FUNCTIONS

Program 2.3 shows a fragment of code that illustrates how class objects are declared and how member functions that operate on these class objects 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 (->), which we will write as →, for indirect component selection through a pointer.

```
// In a source file main.C
#include <iostream.h>
#include "Rectangle.h"

main() {
  Rectangle r, s;     // r and s are objects of class Rectangle
  Rectangle *t = &s;  // t is a pointer to class object s
   .
   .
  // use · to access members of class objects.
  // use → to access members of class objects through pointers.
  if (r.GetHeight () * r.GetWidth () > t→GetHeight () * t→GetWidth ())
    cout << " r ";
  else cout << " s ";
  cout << "has the greater area" << endl;
}
```

Program 2.3: A C++ code fragment demonstrating how *Rectangle* objects are declared and member functions invoked

SPECIAL CLASS OPERATIONS

Constructors and Destructors: The constructor and destructor 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. If a constructor is not defined for a class, memory is allocated for the data members of a class object, when it is created, but the data members are not initialized. The advantage of defining constructors for a class is that all class objects are well-defined as soon as they are created. This eliminates errors that result from accessing an undefined object. A destructor is a member function which deletes data members immediately before the object disappears. A constructor and destructor for class Rectangle are declared in program 2.1.

A constructor must be declared as a public member function of its class; The name of a constructor must be identical to the name of the class to which it belongs; and a constructor must not specify a return type or return a value. Program 2.4 shows a constructor definition for class Rectangle.

```
Rectangle::Rectangle( int x, int y, int height, int width)
{
  x1 = x ; y1 = y ;
  h = height ; w = width ;
}
```

Program 2.4: Definition of a constructor for *Rectangle*

Constructors may be used to initialize Rectangle objects as follows:

```
Rectangle r(1, 3, 6, 6);
Rectangle *s=new Rectangle(0, 0, 3, 4);
```

These create r, a square of side 6 whose bottom left corner is at (1,3); and s, a pointer to a Rectangle object of height 3 and width 4 whose bottom left corner is at the origin.

Destructors are automatically invoked when a class object goes out of scope or when a class object is deleted. Like a constructor, a destructor must be declared as a public member of its class; its name must be identical to the name of its class prefixed with a tilde (~); a destructor must not specify a return type or return a value, and a destructor may not take arguments. If a destructor is not defined for a class, the deletion of an object of that class results in the freeing of memory associated with data members of the class. If a data member is a pointer to some other object, the space allocated to the pointer is returned, but the object that it was pointing to is not deleted. If we also wish to delete this object, we must define a destructor that explicitly does so.

Operator Overloading: Consider the operator == which is used to check for equality between two data items. The operator == may be used to check for equality between two **float** data items; it can also be used to check for equality between two **int** items. The hardware algorithms implementing operator == depend on the type of the operands being compared; that is, the algorithm for comparing two **float**s is different from the one used to compare two **int**s. This is an example of *operator overloading*. However, if we were to try to use operator == to check for equality between two *Rectangle* objects, the compiler would complain that operator == is not defined for *Rectangle* objects. C++ allows the programmer to overload operators for user-defined data types. This is done by providing a definition that implements the operator for the particular data type. This definition takes the form of a class member function or an ordinary function, depending on the operator. The function prototype used must adhere to the specifications for the particular operator. For details about the specifications for various operators, see one of the introductory texts listed at the end of this chapter.

Program 2.6 overloads operator == for **class** *Rectangle*. Our program uses the **this** pointer which we describe briefly: The C++ keyword **this**, when used inside a member function of a class, represents a pointer to the particular class object that invoked it. The class object, itself, is therefore represented by ***this**.

```
int Rectangle::operator== (const Rectangle& s)
{
   if (this == &s) return 1 ;
   if ((x1 == s.x1) && (y1 == s.y1) && (h == s.h)  &&  (w == s.w) ) return 1 ;
   else return 0 ;
}
```

Program 2.6: Overloading **operator==** for **class** *Rectangle*

We can now use the operator == to determine whether two rectangles are identical. Our program first evaluates the expression "this== &s". This expression checks to see if the two rectangles being compared are the same object.

Program 2.7 overloads operator << so that Rectangle objects can be output by using cout.

```
ostream& operator<< (ostream& os, Rectangle& r)
{
   os << "Position is: "<< r.x1 << " " ;
   os << r.y1 << endl ;
   os << "Height is: " << r.h << endl ;
   os << "Width is: " << r.w << endl ;
   return os ;
}
```

Program 2.7: Overloading **operator<<** for **class** *Rectangle*

Notice that operator << accesses private data members of class Rectangle. Therefore, it must be made a friend of Rectangle.

MISCELLANEOUS TOPICS

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 (data or function) has **public**, **private**, or **protected** access, then the member has **public** access. In a class, the default is **private** access. Thus, the C++ **struct** is a generalization of the C **struct**.

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. This is useful in applications where it is known that only one of many possible data items, each of a different type, needs to be stored in a structure; but there is no way to know what that data type is until runtime. The **struct** or **class** structures reserve memory for all their data members. Thus, using a **union** results in a more memory-efficient program, in these cases. We will use **union** in Chapter 4 on linked lists; we will also study a technique for improving on **union** by using inheritance.

A **static** class data member may be thought of as a global variable for its class. From the perspective of a class member function, a static data member is like any other data member. 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. A second difference is that the declaration of a static data member in its class does not constitute a definition. Consequently, a definition of the data member is required somewhere else in the program. We will see an example of a static class member later in this chapter, when we implement the *Polynomial* data type.

ADTs and C++ CLASSES

Note that there is one significant aspect in which the format of ADT differs from the C++ class; some operators in C++ such as operator <<, when overloaded for user-defined ADTs, do not exist as member functions of the corresponding class. Rather, these operators exist as ordinary C++ functions. Thus, these operations are declared outside the C++ class definition of the ADT even though they are actually part of the ADT.

2. DATA STRUCTURE

A ***data structure*** is a systematic way of organizing and accessing data.
Data may be organized in many different ways.
The logical or mathematical model of a particular organization of data is called a Data structure.

A *data structure* tries to structure data

- Usually more than one piece of data
- Should define legal operations on the data
- The data might be grouped together (e.g. in an linked list)

Types of Data Structures:
Data Structures are broadly classified into two types:

1. Linear Data Structure 2. Non Linear Data Structure

1. Linear Data Structures
Definition: A **data structure** is said to be ***linear*** if its elements form a sequence or a **linear** list.
Examples:

- Array
- Linked List
- Stacks
- Queues

2. Non-Linear Data Structures
Examples: Trees, Graphs

Operations on Linear and Non Linear Data Structures

- *Traversal* : Visit every part of the data structure
- *Search* : Traversal through the data structure for a given element
- *Insertion* : Adding new elements to the data structure
- *Deletion:* Removing an element from the data structure.
- *Sorting* : Rearranging the elements in some type of order(e.g Increasing or Decreasing)
- *Merging* : Combining two similar data structures into one

3. Arrays

The simplest type of data structure is an array. Array is a list of finite number of homogeneous data elements such that

a) The elements of array are referred respectively by an index.
b) The elements in an array are stored respectively in successive memory locations.
c) Elements of an array „A" are denoted as A[0], A[1], A[2],……., A[n].

In the notation A[k]
k is called a subscript
A[k] is called variable or subscript value.

Eg: Let „A" be a „6" element array of integers such that A[0]= 247, A[1]= 30, A[2]= 2, A[3]= 200,A[4]=6, A[5]=8

A→

247	30	2	200	6	8
0	1	2	3	4	5

Arrays as ADT
An array is a fundamental abstract data type. Each instance of an array is a set of pairs of the form <index, value>. No two pairs in this set have same index.
Operations: Operations performed on the array are,

1) *Create an array:* This operation creates and initializes an array.
2) *Get an element:* Get's the value of the pair that has a given index.
3) *Set an element:* Adds a pair of the form<index, value> to the array and if a pair with the same index already exists it deletes the old pair.
4) *Insert an element:* Adds a pair of the form<index, value> to the array and if a pair with the same index already exists, move all the elements to the next position (Last to the given index) and element is inserted in given index.
5) *Remove an element:* Deletes an element at given index.
6) *Search an element:* Find's the given element by comparing all elements in an array and if element is found, it returns index of that element. Otherwise, it returns -1.
7) *Display elements:* Displays all elements of an array.

8) *Attributes:* Displays attributes of an array.
9) *Sort:* Sorts all elements in an array.

3.2. ADT Specification of an array

ADT Array

Set of <index, value>
No two pairs have same index

Data Structures

Type *a;
int size;

Operations

void create(s): This operation creates and initializes an array.

void set(i, v): Adds a pair of the form<index, value> to the array and if a pair with the same index already exists it deletes the old pair.

Type get(i): Get's the value of the pair that has a given index.

int search(v): Find's the given element by comparing all elements in an array and if element is found, it returns index of that element. Otherwise, it returns -1.

void display( ): Displays all elements of an array.

End Array

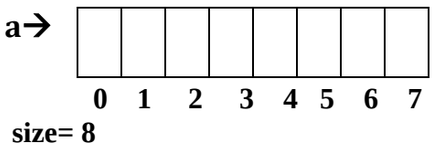
Implementation of an Array ADT using C++

```
template<class T>
class Array
{
        T *a;
        int size;
  public:
        void create(int s);
        void set(int i, T v);
        T get(int i);
        int search(T v);
        void display( );
};
```

i) Creating an array: This operation creates and initializes an array.

```
template<class T>
void Array<T> : : create(int s)
{
        a= new T[s];
        size= s;
}
```

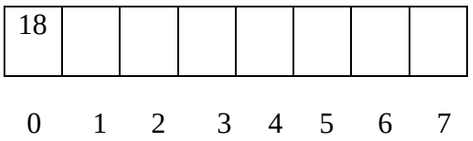
In main, if we call
a1. create(8)



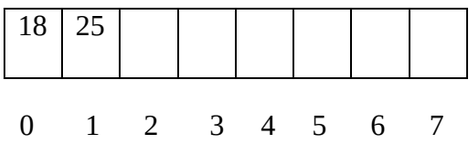
ii) **Set an element:** Adds a pair of the form<index, value> to the array and if a pair with the same index already exists it deletes the old pair.

```
template<class T>
void Array<T> : : set(int i, int v)
{
        if(i< size)
            a[i]= v;
        else
            cout<<"Array out of bound";
}
```

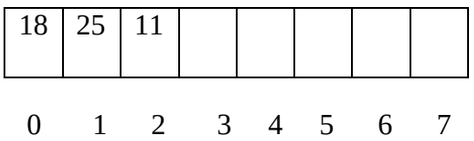
In main, if we call
a1. set(0, 18);



a1. set(1, 25);



a1. set(2, 11);



iii) **Get an element:** Get"s the value of the pair that has a given index.

```
template<class T>
T Array<T>: : get(int i)
{
        return(a[i]);
}
```

In main, if we call,
a1.get(1) ----------25 is the value returned.

iv) **Search an element:** Find"s the given element by comparing all elements in an array and if element is found, it returns index of that element. Otherwise, it returns -1.

```
template<class T>
  int array<T>: : search(T  v)
{
        for( int i=0; i < size; i++)
        if(a[i]==v)
        {
                return i ;
        }
        else
        return  -1
}
```

In main, if we call
a1. search(18);

a→

18	25	11					
0	1	2	3	4	5	6	7

v→ 18
returns **index 0**

v) Display an element: Displays all elements of an array.

```
template<class T>
 void array<T>: : display( )
{
        for( int i=0; i< size; i++ )
        cout<< a[i]<< end l ;
}
```

In main, if we call
a1. display( )

a→

18	25	11					
0	1	2	3	4	5	6	7

Result is 18
25
11

4. Polynomial

Polynomial is a sum of terms, where each has a form a
where "x" is a variable,
"a" is the coefficient,
"e" is the exponent.

Ex: A(x)=3x^20+2x^5+4
B(x)= x^4+10x^3+3x^2+1

The largest exponent of a polynomial is called its degree. Polynomial can be generalized into

$f(x) = \sum a_i x_i$

Coefficient that are zero are not displayed. The terms with exponents equal to zero does not show the variable, since $x^0 = 1$.

Polynomial Representation

Polynomial may be represented using array (or) linked lists.

Polynomial as Array Representation

It is assumed that exponent of a expression are arranged from 0 to highest value(degree) which is represented by subscripts(index) of respective exponents are placed at appropriate index in the array.

Ex: P(x) = 4x^3+6x^2+9

P=

9	0	6	4
0	1	2	3

REPRESENTATION-1:

$P_1(x)$=8x^3+3x^2+2x+6
$P_2(x)$=2x^4+18x-3
$P_3(x)$=16x^21-3x^5+2x+6

P1=

6	2	3	8
0	1	2	3

P2 =

-3	18	0	0	2
0	1	2	3	4

P3

6	2	0	0	0	-3	- - - - - -	16
0	1	2	3	4	5		21

Advantages of Representation1

1. Only good for non spare polynomials.
2. Easy of storage and retrival.

Disadvantages of Representaion1

1. Have to allocate array size ahead of times.
2. Huge array size is required for spares polynomials, waster of space.

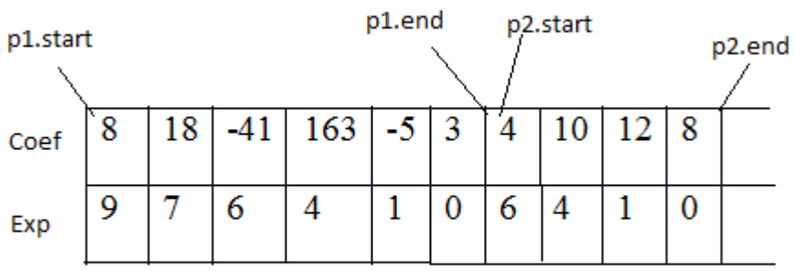
REPRESENTATION -2:

We call this type of representation as double array representation.

Ex:

P1=8x^9+18x^7-41x^6+163x^4-5x+3

P2=4x^6+10x^4+12x+8



Advantages of Representation 2:

1. This type of representation saves space.

Disadvantages of Representation 2:

1. Difficult to maintain.
2. More code required for polynomial operations like addition, subtraction, etc.,

4.3 Operations performed on Polynomials

i) **Create a polynomial:** This function creates dynamic array of coefficient and initializes it to zero.
ii) **Reading a polynomial:** Creates an array dynamically and reads coefficient of all exponents in a polynomial.
iii) **Set a coefficient:** It sets the coefficient in the polynomial at a given exponent (index).
iv) **Get a coefficient:** Returns a coefficient at a given index (or) a exponent.
v) **Degree:** Returns degree of the polynomial.
vi) **Evaluate:** Evaluate the polynomial for a given value of "x".
vii) **Add:** Adds one polynomial to another polynomial.
viii) **Subtract:** Subtracts one polynomial to another polynomial.
ix) **Multiply:** Multiplies two polynomials.
x) **Multiply by constant:** Multiply one polynomial by given constant.
xi) **Equals:** Check equality of two polynomials.
xii) **Derivative:** Computes derivate of a polynomial.
xiii) **Integrate:** Compute integration of a polynomial.

Polynomial as ADT

ADT Polynomial
Sum of < a_i, e_i >
Where a_i ,s are coefficients and e_i,are exponents
Data structures
Type *coefficient;
int deg;
Operations
void createpoly (int d)*:* create dynamic array & construct zero polynomial.
void readpoly (int d)*:* creates dynamic array & reads coefficient from user.
void setcoeff (Type coef , int ex)*:* sets a coefficient at a given exponent (index).
void getcoeff (int ex)*:* Retrives coefficient at a given index.
int degree ( ): retrieves degree of a polynomial.
Type evaluate (int x): Evaluate a polynomial by given x.
Polynomial addpoly(polynomial A, polynomial B)*:* add two polynomials.
Polynomial subpoly(polynomial A, polynomial B): subtract two polynomials
Polynomial mulpoly(polynomial A, polynomial B): multiply two polynomials
void displaypoly( ):displays polynomial such that zero coefficients aren‟t displayed.
End polynomial.

Polynomial implementation using C++

```
template <class t>
  Class Poly
  {
            T *coef;
            int deg;
      public:
            void createpoly(int d);
            void readpoly(int d);
            void setcoeff(T c, int e);
            T getcoeff (int e);
            Poly addpoly(Poly A, Poly B);
            void display ( );
   };
```

i) **Creating a polynomial:** This function creates a dynamic array of coefficient of size degree +1 and creates zero polynomial.

```
template <class T>
 void Poly <T>::createpoly (int d)
 {
        coef=new T[d+1];
        deg=d;
        for(int i=0;i<=deg; i++)
              coef[i]=0;
}
```

In main function; we call
p.createpoly (4);

coef

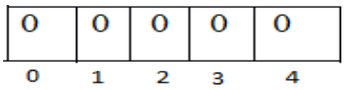
0	0	0	0	0
0	1	2	3	4

deg=4

ii) **Reading a polynomial:** This function reads a coefficient from user and place in the coef array.

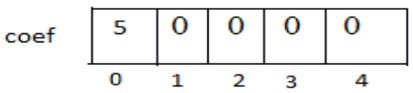
```
template <class T>
        void Poly<T>::readpoly( )
        {
                for(int i=0; i<=deg; i++)
                {
                        cout <<"enter coef for exponent"<< i;
                        cin >>coef[i];
                }
        }
```

In main
p.readpoly( );

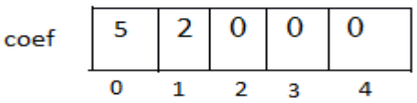


then,

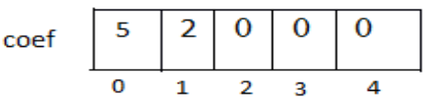
at i=0; enter coef for exponent 0
5



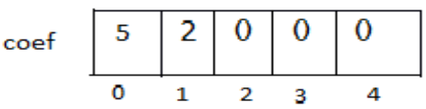
at i=1; enter coef for exponent 1
2



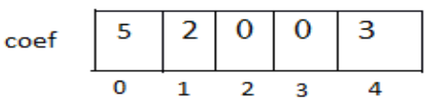
at i=2; enter coef for exponent2
0



at i=3; enter coef for exponent 3
0



at i=4; enter coef for exponent4
3



iii) **Displaying a polynomial:** This function displays the polynomial such that zero coefficient terms are not displayed.

```
template <class T>
 void Poly <T>::display( )
{
        for(int i=deg; i>=0; i--)
        {
                if (coef [i] !=0)
                {
                        cout<<coef[i]<<"X^"<<i;
                        if (i!=0)
                        cout<<"+";

                }
         }
}
```

For above example, the output produced is as follows

At i=4 ------> 3 x ^4 +
At i=3 ------ > 3 x ^4 +
At i=2 ------ > 3 x ^4 +
At i=1 ------ > 3 x ^4 + 2 x ^1+
At i=0 ------ > 3 x ^4 + 2 x ^1+ 5

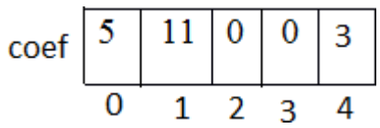
iv) **Set a coefficient:** This function sets coefficient of a given exponent value.

```
template <class T>
void Poly<T>::setcoef(T c, int e)
{
        coef [e]=c;
}
```

In main, if we call

p.set (11,1)

then, coef[1] is set to 11;



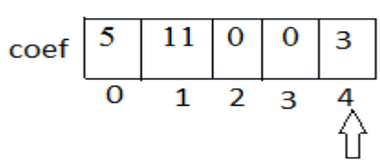
v) **Get coefficient:** This function retrives a coefficient at a given index (or) exponent.

```
template <classT>
T poly<T>::getcoef(int e)
{
        return coef[e];
}
```

In main, if we call

p.getcoef(4);

then, 3 will be returned (Since, coef [4] is 3)



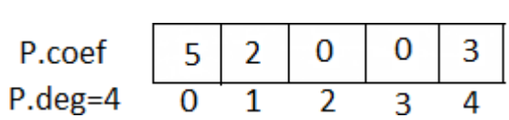
vi) **Addition of two Polynomials:** This function adds two polynomial and returns the sum of two polynomial.

```
template <class T>
Poly Poly<T>::addpoly(Poly A, Poly B)
{
        if (A.deg>B.deg)
                deg=A.deg;
        else
                deg=B.deg;

        createpoly(deg);

        for(int i=0;i<=deg;i++)
                coef[i]=A.coef[i]+B.coef[i];
        return  *this;
}
```

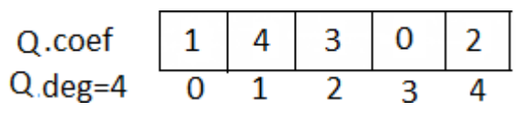
Let us consider two polynomial P & Q

P=3x^4+2x+5 Q=2x^4+2x^3+4x+1

P can be represented as



Q can be represented as



In main function , if we call

R.addploy(P,Q) then P,Q polynomials are copied to A&B, and the sum of two polynomials is as follows:

Now, R.deg=max (P.degree, Q.degree)=4

R.coef	6	6	3	0	5
R.deg =4	0	1	2	3	4

5. SPARSE MATRICES

A sparse matrix is a matrix having relatively small number of non-zero elements. Sparse matrix is a 2D array in which most of the elements have null value or zero. It is wastage of memory and processing time if we store null value of matrix in array.

A sparse matrix is matrix in which number of zero elements are more than number of non-zero elements.

Ex: Diagonal Matrices, Lower triangular matrices etc.,

2 3 0
0 0 0
0 1 0 is sparse matrix.

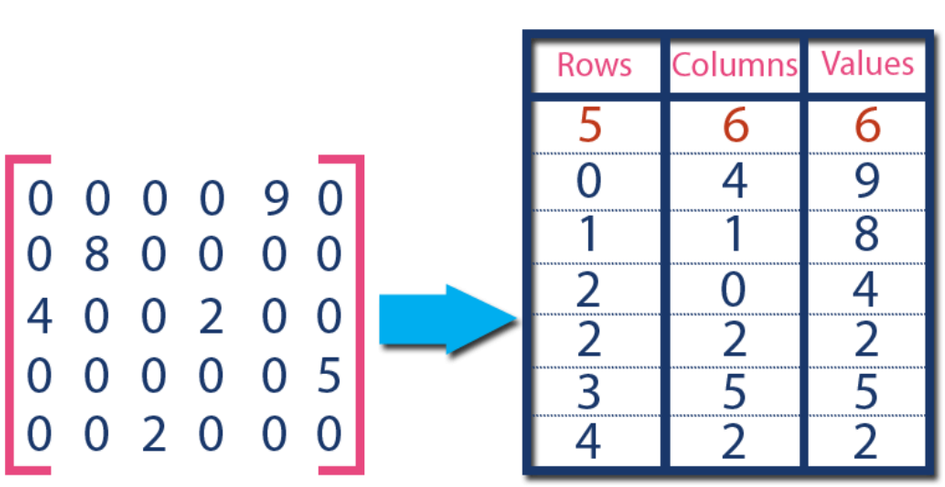
Sparse Matrix can be represented using Triplet and Linked List.

Representation using Triplet

Triplet Representation

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0^{th} row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non- zero values. This matrix can be represented as shown in the image...



In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which

indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

Let us consider another sparse matrix

2 3 0
0 0 0
0 -5 0

Sparse matrix can be represented using array of triples<row, col, value>. Scan non-zero elements of Sparse Matrix in row major order. Each non-zero element is represented by triple<row, col, value>. The following is the sparse matrix representation:

	row	col	value
t[0]	3	3	3
t[1]	0	0	2
t[2]	0	1	3
t[3]	2	1	-5

Thus, t[0].row contains max no. of rows; t[0].col contains max. no. columns; t[0].value contains total no. of non-zero elements. Positions 1 to 3 store the triples representing non-zero entries. The row index is in the field row; the column index is in the field col; and the value is in the field value.

```
class Term
{
        public:
                int row;
                int col;
                int value;
};
class SparseMatrix
{
       Term t[20];
       ……
};
```

Here each term is a triple <row,col,value>.

Sparse Matrix ADT

ADT SparseMatrix

A set of triples, <row, col, value>, where row and col are integers and form a unique combination.

Data Structures

Term t[10]; -Array of Terms where Term is <row, col, val>

Operations

void create(n): creates a SparseMatrix that can hold n non-zero elements information.

SparseMatrix transpose(A): return the matrix produced by interchanging the row and column value of every triple.

SparseMatrix add(A,B): if dimensions of a and b are the same return the matrix produced by corresponding items, namely those with identical row and column values else return error.

SparseMatrix multiply(A,B): if number of columns in a equals number of rows in B return the matrix D produced by multiplying A by B according to the formula: D[i][j]= Σ(A[i][k]*B[k][j]) where D[i][j] is the (i,j) th element else return error.

End SparseMatrix

Sparse Matrix Transpose

To transpose a matrix we must interchange the rows and columns.

Example: matrix A

	0	1	2	3	5	6
0|	15	0	0	22	0	-15
1|	0	11	3	0	0	0
2|	0	0	0	-6	0	0
3|	0	0	0	0	0	0
4|	91	0	0	0	0	0
5|	0	0	28	0	0	0

	row	col	value
A.t[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

	row	col	value
B.t[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

For instance, in the above example

(0,0,15)	which becomes	(0,0,15)
(0,3,22)	which becomes	(3,0,22)
(0,5,-15)	which becomes	(5,0,-15)

If we place these triples consecutively in the transpose matrix, then as we insert new triples, we must move elements to maintain the correct order. We can avoid this data movement by using the column indices to indices to determine the placement of elements in the transpose matrix.

The algorithm indicates that we should “find all the elements in column 0 and store them in row 0 of the transpose matrix, find all the elements in column 1 and store them in row 1 etc.”

Implementation of Transpose of Sparse Matrix

```
template <class T>
SparseMatrix SparseMatrix<T>::transpose(SparseMatrix A)
{
        t[0].row=A.t[0].row;
        t[0].col=A.t[0].col;
        t[0].value=A.t[0].value;

        n=t[0].value;
        k=1;
        if(n>0)
        {
                for(i=0; i<A.t[0].col; i++)
                  for(j=0; j<=n; j++)
                        {
                                t[k].row=A.t[j].col;
                                t[k].col=A.t[j].row;
                                t[k].value=A.t[j].vlaue;
                                k++;
                        }
        }
         return *this;
}
```

Analysis of transpose: Determining the algorithm computing time of this algorithm is easy since the nested for loops are the decisive factor. We can see that the outer for loop is iterated A.t[0].col times(no. of columns in the original matrix). One iteration of the inner for loop requires A.t[0].value (no. of non-zero elements in the original matrix). Therefore, the total time for the nested for loops is columns * elements. Hence, the asymptotic time complexity is O(columns*elements).Much better algorithm can be created by using a little storage, in which

we can transpose a matrix represented as a sequence of triples in O(columns + elements) time.

Matrix multiplication
Definition:
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

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

for $0 \le i < m$ and $0 \le j < p$.

Example:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 2.5: Multiplication of two sparse matrices

Sparse Matrix Multiplication
Definition: $[D]_{m*p}=[A]_{m*n} * [B]_{n*p}$
Procedure: Fix a row of A and find all elements in column *j* of *B* for *j*=0, 1, …, *p*-1.

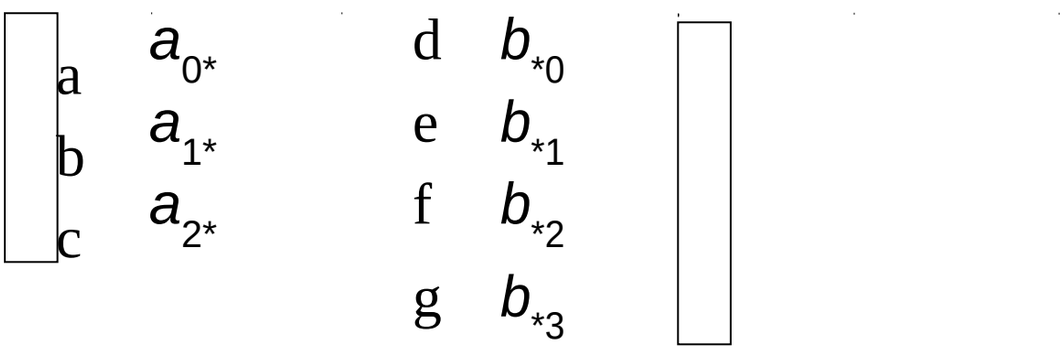
Alternative 1.

Scan all of *B* to find all elements in *j*.

Alternative 2.

- Compute the transpose of *B*. (Put all column elements consecutively)
- Once we have located the elements of row *i* of *A* and column *j* of *B* we just do a merge operation similar to that used in the polynomial addition

General case:
$d_{ij}=a_{i0}*b_{0j}+a_{i1}*b_{1j}+\ldots+a_{i(n-1)}*b_{(n-1)j}$
Array A is grouped by i, and after transpose, array B is also grouped by j.



The multiply operation generate entries:
a*d , a*e , a*f , a*g , b*d , b*e , b*f , b*g , c*d , c*e , c*f , c*g

The below program can obtain the product matrix *D* which multiplies matrices *A* and *B*.

```
void mmult(term a[], term b[], term d[])
/* multiply two sparse matrices */
{
   int i, j, column, totalb = b[0].value, totald = 0;
   int rows_a = a[0].row, cols_a = a[0].col,
   totala = a[0].value; int  cols_b = b[0].col,
   int row_begin = 1, row = a[1].row, sum = 0;
   int new_b[MAX_TERMS][3];
   if (cols_a != b[0].row) {
      fprintf(stderr,"Incompatible matrices\n");
      exit(1);
   }
   fast_transpose(b,new_b);
   /* set boundary condition */
   a[totala+1].row = rows_a;
   new_b[totalb+1].row = cols_b;
   new_b[totalb+1].col = 0;
   for (i = 1; i <= totala; ) {
      column = new_b[1].row;
      for (j = 1; j <= totalb+1;) {
      /* multiply row of a by column of b */
        if (a[i].row != row) {
           storesum(d,&totald,row,column,&sum);
           i = row_begin;
           for (; new_b[j].row == column; j++)
              ;
           column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
           storesum(d, &totald, row, column, &sum);
           i = row_begin;
           column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[j].col)) {
           case -1: /* go to next term in a */
                i++;  break;
           case 0: /* add terms, go to next term in a and b*/
                sum += ( a[i++].value * new_b[j++].value);
                break;
           case 1 : /* advance to next term in b */
                j++;
        }
      }  /* end of for j <= totalb+1 */
      for (; a[i].row == row; i++)
        ;
      row_begin = i; row = a[i].row;
   } /* end of for i<=totala */
   d[0].row = rows_a;
   d[0].col = cols_b; d[0].value =  totald;
}
```

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 $A[i_1][i_2],...,[i_n]$, gets mapped onto a position in a one-dimensional C++ array so that it can be retrieved efficiently . This is necessary since programs using arrays may, in general, use array elements in a random order. In addition to being able to retrieve array elements easily, it is also necessary to be able to determine the amount of memory space to be reserved for a particular array. Assuming that each array element requires only one word of memory, the number of words needed is the number of elements in the array. If an array is declared $A[p_1 .. q_1][p_2 .. q_2], \cdots, [p_n .. q_n]$, where $p_i .. q_i$ is the range of index values in dimension i, then it is easy to see that the number of elements is

$$\prod_{i=1}^{n} (q_i - p_i + 1)$$

One of the common ways to represent an array is in *row major order* (see Exercise 4 at the end of this section for column major order). If we have the declaration

A[4..5][2..4][1..2][3..4]

then we have a total of 2*3*2*2 = 24 elements. Using row major order, these elements will be stored as

A[4][2][1][3], A[4][2][1][4], A[4][2][2][3], A[4][2][2][4]

and continuing

A[4][3][1][3], A[4][3][1][4], A[4][3][2][3], A[4][3][2][4]

for three more sets of four until we get

A[5][4][1][3], A[5][4][1][4], A[5][4][2][3], A[5][4][2][4]

We see that the index at the right moves the fastest. In fact, if we view the indices as numbers, we see that they are, in some sense, increasing:

4213, 4214, $\cdots$, 5423, 5424

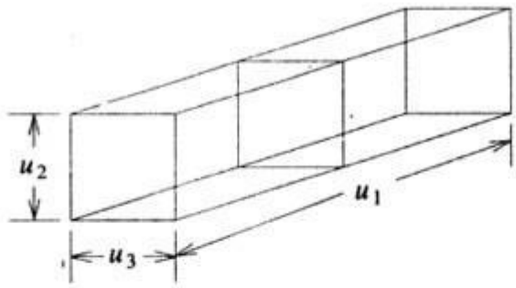
A synonym for row major order is *lexicographic order*.

The problem is how to translate from the name $A[i_1][i_2], ..., [i_n]$ to the correct location in the one-dimensional array. Suppose A[4][2][1][3] is stored at position 0. Then A[4][2][1][4] will be at position 1 and A[5][4][2][4] at position 23. These two addresses are easy to guess. In general, we can derive a formula for the address of any element. This formula makes use of only the starting address of the array plus the declared dimensions.

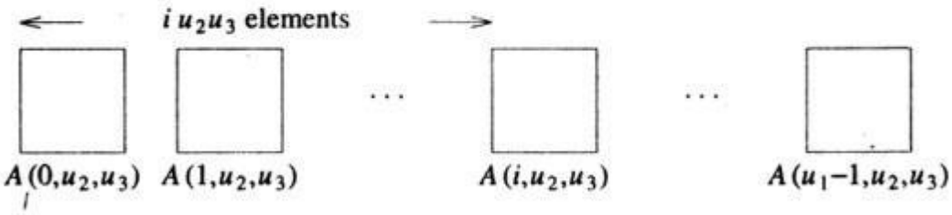
To simplify the discussion we shall assume that indices in dimension i run from 0 to $u_i - 1$ (that is, $p_i = 0$ and $q_i = u_i - 1$). The general case when p_i can be any integer is discussed in the exercises. Before obtaining a formula for the case of an n-dimensional array, let us look at the row major representation of one-, two-, and three-dimensional arrays. To begin with, if A is declared $A[u_1]$, then assuming one word per element, it may be represented in sequential memory as in Figure 2.5. If α is the address of $A[0]$, then the address of an arbitrary element $A[i]$ is just $\alpha + i$.

array element:	$A[0]$	$A[1]$	$A[2]$	$\cdots$	$A[i]$	$\cdots$	$A[u_1-1]$
address:	α	$\alpha+1$	$\alpha+2$	$\cdots$	$\alpha+i$	$\cdots$	$\alpha+u_1-1$

Figure 2.5: Sequential representation of $A[u_1]$



(a) 3-dimensional array $A[u_1][u_2][u_3]$ regarded as u_1 2-dimensional arrays



(b) Sequential row major representation of a 3-dimensional array. Each 2-dimensional array is represented as in Figure 2.6

Figure 2.7: Sequential representation of $A[u_1][u_2][u_3]$