Problem Statement:

Write a program to implement a Diagonal Matrix a Lower Triangular Matrix, an Upper Triangular Matrix, and a Symmetric Matrix using a one-dimensional array.

Problem Definition:

Matrix memory optimization means storing special matrices efficiently by avoiding storage of predictable zero or duplicate elements, using compact 1D array instead of full 2D array.

- **Diagonal Matrix:** Only diagonal elements `(i == j)` are non-zero. Stored using the formula `arr[i]`.
- **Lower Triangular Matrix:** All elements above the main diagonal are zero. Non-zero elements mapped using `i*(i+1)/2 + j`.
- **Upper Triangular Matrix:** All elements below the main diagonal are zero. Non-zero elements mapped using `n*i - i*(i-1)/2 + (j-i)`.
- **Symmetric Matrix:** Matrix equal to its transpose `(A[i][j] = A[j][i])`. Only the lower triangular half is stored, using `i*(i+1)/2 + j`.

Algorithm:

Diagonal Matrix:

```
procedure setDiagonal(arr, i, j, x)
begin
	if(i == j)
		arr[i] ← x;
	end if
end procedure

procedure getDiagonal(arr, i, j)
begin
		if(i == j)
		return arr[i];
	else
		return 0;
	end if
end procedure
```

Lower Triangular Matrix:

```
procedure setLower(arr, i, j, x)
begin
	if(i ⩾ j)
		arr[i*(i+1)/2+j] ← x;
	end if
end procedure

procedure getLower(arr, i, j)
begin
	if(i ⩾ j)
		return arr[i*(i+1)/2+j];
	else
		return 0;
	end if
end procedure
```

Upper Triangular Matrix:

```
Procedure setUpper(arr, n, i, j, x)
begin
       if(i ⩽ j)
              arr[n*i-i*(i-1)/2+(j-i)] ← x;
       end if
end procedure


Procedure getUpper(arr, n, i, j)
begin
       if(i ⩽ j)
              return arr[n*i-i*(i-1)/2+(j-i)];
       else
              return 0;
       end if
end procedure
```

Symmetric Matrix:

```
Procedure setSymmetric(arr, i, j, x)
begin
       if(i ⩾ j)
              arr[i*(i+1)/2+j] ← x;
       else
              arr[j*(j+1)/2+i] ← x;
       end if
end procedure

Procedure getSymmetric(arr, i, j)
begin
       if(i ⩾ j)
              return arr[i*(i+1)/2+j];
       else
              return arr[j*(j+1)/2+i];
       end if
end procedure
```

Source Code:

```
#include <iostream>

using namespace std;

class Matrix
{
private:
  int n, *arr;

public:
  Matrix(int size)
  {
    n = size;
    arr = new int[n * (n + 1) / 2];
  }

  void setDiagonal(int i, int j, int x)
  {
    if (i == j)
      arr[i] = x;
  }
  void setLowerTri(int i, int j, int x)
  {
    if (i >= j)
      arr[i * (i + 1) / 2 + j] = x;
  }
  void setUpperTri(int i, int j, int x)
  {
    if (i <= j)
      arr[n * i - (i * (i - 1) / 2) + (j - i)] = x;
  }
  void setSymmetric(int i, int j, int x)
  {
    (i >= j) ? arr[i * (i + 1) / 2 + j] = x : arr[j * (j + 1) / 2 + i] = x;
  }

  int getDiagonal(int i, int j)
  {
    return (i == j) ? arr[i] : 0;
  }
  int getLowerTri(int i, int j)
  {
    return (i >= j) ? arr[i * (i + 1) / 2 + j] : 0;
  }
  int getUpperTri(int i, int j)
  {
    return (i <= j) ? arr[n * i - i * (i - 1) / 2 + (j - i)] : 0;
  }
  int getSymmetric(int i, int j)
  {
    return (i >= j) ? arr[i * (i + 1) / 2 + j] : arr[j * (j + 1) / 2 + i];
  }

  void display(int type)
  {
    cout << endl
         << "Matrix:" << endl;
    for (int i = 0; i < n; i++)
    {
      for (int j = 0; j < n; j++)
```

```
      {
        if (type == 1)
          cout << getDiagonal(i, j) << " ";
        else if (type == 2)
          cout << getLowerTri(i, j) << " ";
        else if (type == 3)
          cout << getUpperTri(i, j) << " ";
        else if (type == 4)
          cout << getSymmetric(i, j) << " ";
      }
      cout << endl;
    }
  }

  ~Matrix()
  {
    delete[] arr;
  }
};

int main()
{
  int n, choice, x;
  cout << "Enter the order of the matrix : ";
  cin >> n;
  Matrix M(n);

  cout << "\n1. Diagonal Matrix.";
  cout << "\n2. Lower Triangular Matrix.";
  cout << "\n3. Upper Triangular Matrix.";
  cout << "\n4. Symmetric Matrix.";
  cout << "\nEnter your choice : ";
  cin >> choice;

  switch (choice)
  {
  case 1:
    cout << "\nEnter the diagonal elements : ";
    for (int i = 0; i < n; i++)
    {
      cin >> x;
      M.setDiagonal(i, i, x);
    }
    M.display(1);
    break;

  case 2:
    cout << "\nEnter the elements : ";
    for (int i = 0; i < n; i++)
    {
      for (int j = 0; j ≤ i; j++)
      {
        cin >> x;
        M.setLowerTri(i, j, x);
      }
    }
    M.display(2);
    break;

  case 3:
    cout << "\nEnter the elements : ";
    for (int i = 0; i < n; i++)
```

```
    {
      for (int j = i; j < n; j++)
      {
        cin >> x;
        M.setUpperTri(i, j, x);
      }
    }
    M.display(3);
    break;

  case 4:
    cout << "\nEnter the elements : ";
    for (int i = 0; i < n; i++)
    {
      for (int j = 0; j ⩽ i; j++)
      {
        cin >> x;
        M.setSymmetric(i, j, x);
      }
    }
    M.display(4);
    break;

  default:
    cout << "Invalid Choice.";
  }
  return 0;
}
```

Output:

```
notamitgamer@fedora:~/Documents/bsc/semester_2/assignments$ ./a.out
Enter the order of the matrix : 3

1. Diagonal Matrix.
2. Lower Triangular Matrix.
3. Upper Triangular Matrix.
4. Symmetric Matrix.
Enter your choice : 1

Enter the diagonal elements : 1 2 3

Matrix:
1 0 0
0 2 0
0 0 3
notamitgamer@fedora:~/Documents/bsc/semester_2/assignments$ ./a.out
Enter the order of the matrix : 3

1. Diagonal Matrix.
2. Lower Triangular Matrix.
3. Upper Triangular Matrix.
4. Symmetric Matrix.
Enter your choice : 2

Enter the elements : 1 2 3 4 5 6

Matrix:
1 0 0
2 3 0
4 5 6
```

```
notamitgamer@fedora:~/Documents/bsc/semester_2/assignments$ ./a.out
Enter the order of the matrix : 3

1. Diagonal Matrix.
2. Lower Triangular Matrix.
3. Upper Triangular Matrix.
4. Symmetric Matrix.
Enter your choice : 3

Enter the elements : 1 2 3 4 5 6

Matrix:
1 2 3
0 4 5
0 0 6
notamitgamer@fedora:~/Documents/bsc/semester_2/assignments$ ./a.out
Enter the order of the matrix : 3

1. Diagonal Matrix.
2. Lower Triangular Matrix.
3. Upper Triangular Matrix.
4. Symmetric Matrix.
Enter your choice : 4

Enter the elements : 1 2 3 4 5 6

Matrix:
1 2 4
2 3 5
4 5 6
```

P.T.O.

Discussion:

This program demonstrates how memory can be saved when working with special types of matrices — **Diagonal**, **Lower Triangular**, **Upper Triangular**, and **Symmetric** — by storing only their essential elements in a single dynamically allocated 1D array, rather than a full 2D array.

A **`Matrix`** class is used to bundle together the storage (the array) and the logic needed to interpret it correctly for each matrix type.

Two private members drive the class:

- **`arr`** - a dynamically allocated integer pointer that holds the compact array of stored values.
- **`n`** - the order (size) of the square matrix.

When a **`Matrix`** object is created, the constructor computes the array size as **`n*(n+1)/2`** and allocates it on the heap using **`new`**.

The class's functionality is built around these key methods:

- **`setDiagonal()`**, **`setLowerTri()`**, **`setUpperTri()`** and **`setSymmetric()`** – Each of these accepts a row index, column index, and a value, then applies a matrix-specific formula to determine exactly where that value should be placed in the 1D array.

- **`getDiagonal()`**, **`getLowerTri()`**, **`getUpperTri()`** and **`getSymmetric()`** – These reconstruct values for any **`(i, j)`** position by locating them in the array (using ternary conditions), returning 0 for positions that were never stored, and — in the Symmetric case — reflecting values across the diagonal when needed.

- **`display()`** – Using two nested loops, this method walks through every position of the conceptual n×n matrix and prints the corresponding value, effectively rebuilding the full matrix view from the compact array.

Finally, the destructor **`~Matrix()`** ensures proper cleanup by releasing the dynamically allocated array with **`delete[]`** once the object goes out of scope, preventing memory leaks.