Matrices

This C++ file is a masterclass in the **memory-efficient representation of sparse matrices**—matrices that contain a large number of zero elements. Instead of storing a full N x N grid, which would be wasteful, the code demonstrates how to use a single, compact 1D array to store only the non-zero values.

The core principle showcased is the use of **mathematical mapping formulas**. For each type of sparse matrix, a unique formula is used to convert the traditional 2D coordinates (row, column) into a single index for the 1D array. This technique dramatically reduces memory consumption.

The file explores several common types of sparse matrices:

* **Diagonal:** Only the main diagonal has non-zero elements.
* **Lower & Upper Triangular:** Non-zero elements are only on or below/above the main diagonal.
* **Tri-Diagonal:** Non-zero elements are only on the main, lower, and upper diagonals.
* **Toeplitz:** All elements along any given diagonal are identical.

For some of these types, the file provides both a procedural C-style implementation using struct and a more robust, object-oriented C++ implementation using a class. This effectively contrasts two different programming paradigms for solving the same problem.

Finally, the file culminates in a **menu-driven program** that consolidates all these concepts, allowing a user to interactively choose a matrix type, enter its non-zero elements, and see the full N x N matrix displayed correctly, all while being stored efficiently in the background.

**Diagonal matrix (C & C++)**

This code implements a **Diagonal Matrix**, where non-zero elements can only exist on the main diagonal (where the row index equals the column index). Instead of storing all N² elements, this approach uses a 1D array of just N elements to save memory.

* if(i==j){ m->A[i-1] = x; } This is the core logic. The condition i==j ensures that we only care about elements on the main diagonal. The 2D index (i, j) is **mapped** to the 1D array index i-1. For example, element (3, 3) in the 2D matrix is stored at index 2 in our 1D array.
* class matrix{ ... } The C++ version encapsulates this logic within a class. The constructor matrix(int n) handles memory allocation (new int[n]), and the destructor ~matrix() handles deallocation (delete []A), making memory management safer and more automatic than the C-style struct.

**Lower Triangular matrix (C & C++)**

This implements a **Lower Triangular Matrix**, where all elements *above* the main diagonal are zero. The code efficiently stores only the non-zero elements (on and below the diagonal) using a **row-major mapping formula**.

* m->A[i\*(i-1)/2 + j-1] = x; This is the **row-major formula** for mapping 2D indices to a 1D array. It works by calculating the total number of elements in all the full rows *before* the current row i (which is i\*(i-1)/2) and then adding the position of the element within the current row (j-1).
* m.A = (int \*)malloc((m.n\*(m.n+1)/2)\*sizeof(int)); This line allocates exactly the amount of memory needed. The number of non-zero elements in an N x N lower triangular matrix is the sum of integers from 1 to N, which is given by the formula n\*(n+1)/2.

**Tri-Diagonal matrix**

This code implements a **Tri-Diagonal Matrix**, where non-zero elements are restricted to three diagonals: the main diagonal, the lower diagonal (just below the main), and the upper diagonal (just above the main).

* m.A = (int \*)malloc((3\*m.n-2)\*sizeof(int)); An N x N tri-diagonal matrix has n elements on the main diagonal, n-1 on the lower, and n-1 on the upper, for a total of 3n-2 non-zero elements. This allocation is perfectly sized to save memory.
* if(i-j==1){ ... m.A[i-1] } This condition i-j==1 identifies the **lower diagonal**. These n-1 elements are stored first in our 1D array, using the mapping i-1.
* else if(i-j==0){ ... m.A[m.n-1+i-1] } This i-j==0 identifies the **main diagonal**. These n elements are stored *after* the lower diagonal elements. Their mapping is (n-1) + (i-1).
* else if(i-j==-1){ ... m.A[2\*(m.n)-1+i-1] } This i-j==-1 identifies the **upper diagonal**. These n-1 elements are stored last. Their mapping is (n-1) + n + (i-1), which simplifies to 2n-2+i.

**Toeplitz matrix**

This implements a **Toeplitz Matrix**, a special matrix where every descending diagonal from left to right has constant values. This property means the entire N x N matrix can be defined by just its first row and first column.

* m.A = (int \*)malloc((2\*m.n-1)\*sizeof(int)); Since the matrix is defined by the first row (N elements) and first column (N elements), with the top-left element A[0][0] being part of both, we only need to store N + N - 1 = 2N - 1 unique values.
* if (i <= j) { ... m.A[j - i] } For any element on or above the main diagonal (i <= j), its value is determined by an element in the **first row**. The specific element is found by the index j - i, which remains constant along any diagonal.
* else { ... m.A[m.n + i - j - 1] } For any element below the main diagonal (i > j), its value is determined by an element in the **first column**. These values are stored in our 1D array *after* the first row's values. The mapping is n + (i - j - 1).

**Menu driven program for matrices**

This final section of the code brings all the previous concepts together into a single, interactive application. It acts as a user interface for creating and displaying the different types of sparse matrices.

* do{ ... }while(ch!=6); This loop structure ensures the **menu runs continuously**, allowing the user to create multiple matrices without restarting the program. It only exits when the user selects option 6.
* switch (ch){ ... } The switch statement is the control center of the program. Based on the user's input (ch), it executes the correct block of code to handle the creation and display of the chosen matrix type.
* case 3: ... m.A[j\*(j-1)/2 + i-1] = num; This case handles the **Upper Triangular matrix**. It's similar to the lower triangular matrix but uses a **column-major mapping formula**. Here, the roles of i and j are swapped in the formula, effectively storing the non-zero elements column by column instead of row by row.