Question 2

1. Background

Matrices, usually two-dimensional tables of scalar values, are an important tool in mathematics,engineering and science. You will be constructing a C++ class that is able to encode a matrix, together with a series of matrix operations. The horizontal lines of a matrix are called rows and the vertical lines are called *columns*. A matrix with *m* rows and *n* columns is referred to as an *m-by-n(m x n)* matrix. The entry at row *i* and column *j* is the *(i, j)-th* entry. For example, a 4×3 matrix is:

$$\mathbf{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \\ a_{3,0} & a_{3,1} & a_{3,2} \end{bmatrix}$$

Note that matrix rows and columns are zero-indexed. Various operations can be performed on matrices. Addition and subtraction, defined for matrices of identical dimensions, is achieved by element-wise addition and subtraction respectively; multiplication, defined for an *m*×*n* matrix multiplied with an *n*×*p* matrix, is achieved via row-by-column multiplication; transposition, achieved by turning rows into columns and vice-versa, and more...

Sparse matrices are matrices with relatively few non-zero elements (as opposed to 'regular', dense matrices that have few zero elements). Working with complex partial differential equations typically involves very large sparse matrices. It quickly becomes impractical to store such matrices in a dense format, e.g., a 10,000 × 10,000 matrix of integers would require ~400MB of space. Instead, you will use a much more compact sparse representation, together with appropriately optimised operations.

Such a sparse matrix class might appear in a C++ high-performance mathematics library.

2. The Task

You are required to write a C++ class for sparse matrices that operates on any non-floating numeric type.

The specification is provided below. Functions that have friend alternatives, or const-qualified alternatives, are omitted from the specification due to their semantic similarity. However, all are required to be provided.

2.1. Mandatory Internal Representation

You are **required** to use the following representation for your sparse storage format for an *m* x *n* matrix with *k* non-zero entries.

1. A pointer, **vals_**, to an array (with *k* elements in it) of `int` representing the non-zero matrix entries
2. A pointer, **cidx_**, to an array (with *k* elements in it) of `std::size_t` representing the matrix column indices corresponding to each of the entries in **vals_**
3. A map, **ridx_**, of row indices to `pair`s whose first element is an index into **cidx_** corresponding to the first non-zero element in each matrix row and whose second element is the non-zero count of non-zero entries in that row.

These three data members have been provided to you already in `include/q2/q2.hpp`.

Consider the following 8 x 7 matrix with 15 non-zero entries.

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 & 0 & 3 \\ 4 & 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 8 & 0 \\ 9 & 0 & 0 & 10 & 11 & 12 & 0 \\ 0 & 13 & 0 & 0 & 14 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 15 \end{bmatrix}$$

We would encode it as:

$$\begin{aligned} \texttt{vals} &\rightarrow \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\} \\ \texttt{cidx} &\rightarrow \{0, 3, 6, 0, 1, 1, 2, 5, 0, 3, 4, 5, 1, 4, 6\} \\ \texttt{ridx} &\rightarrow \left\{ \begin{array}{lcl} 0 & \mapsto & (0, 3) \\ 1 & \mapsto & (3, 2) \\ 2 & \mapsto & (5, 3) \\ 3 & \mapsto & (8, 4) \\ 4 & \mapsto & (12, 2) \\ 7 & \mapsto & (14, 1) \end{array} \right\} \end{aligned}$$

It is not necessary in general to keep the range of values in `vals_` and `cidx_` (corresponding to each row) ordered; however, you are required to keep them in increasing column-index order for the purposes of this question.

Once you read the interface requirements, you will notice that, while the size of the matrix is fixed upon construction, the number of non-zero values may fluctuate, requiring possible resizing of the representation arrays. You should use the following simple amortisation technique, assuming that you are working with an m × n matrix:

- The `vals_` and `cidx_` arrays should be sized to min((m × n) ÷ 5, 1000) upon construction.

(with the constructor that only specifies an initial matrix size)
- The `vals_` and `cidx_` arrays should be doubled in size upon reaching capacity.

Intuitively, we assume that only a small number of matrix entries will be non-zero and resize as necessary. Better amortisation techniques exist, but this suffices for our purposes.

When non-zero matrix values are set to zero (i.e., via element()) you are required to reorganise the internal representation to remove the new zero value from `vals_` and cidx, and update `ridx_` as appropriate (this includes potentially removing an entry from `ridx_` if a row no longer has any non-zero entries). However, you are not required to downsize the representation arrays.

You absolutely must use dynamically allocated arrays to store the matrix representation. Therefore, the compiler-generated copy-control members will not provide the correct semantics. You must provide your own implementations for these copy-control member functions.

A static data member `matrix_count_` should exist on your matrix. This data member value is increased every time a new matrix is constructed (excluding move construction). This data member is decreased every time a matrix is destructed.

Failure to follow the prescribed internal representation for your matrix will result in a total mark of 0 for this question.

2.2 Constructors

For all constructors, you may assume that the arguments supplied by the user are correct for the given type.

One Dimension

```
sparse_matrix(std::size_t dim = 1);
```

Example: Given `sparse_matrix(3)`, a sparse matrix is constructed as a square matrix of size 3 x 3, with its entries being 0.

Separate Dimensions

```
sparse_matrix(std::size_t rows, std::size_t columns);
```

Example: Given `sparse_matrix(3, 5)`, a 3 x 5 matrix (3 rows, 5 columns) is constructed with entries being 0.

Input stream

```
sparse_matrix(std::istream& is);
```

A sparse matrix is constructed when given a serialized representation read from the input stream which has the same format as the output operator's format.

Copy construction and assignment

```
sparse_matrix(sparse_matrix const& other);
auto operator=(sparse_matrix const& other) -> sparse_matrix&;
```

Implement copy semantics.

Move construction and assignment

`sparse_matrix(sparse_matrix&& other) noexcept;`
`auto sparse_matrix(sparse_matrix&& other) noexcept -> sparse_matrix&;`

Implement move semantics.

2.3 Destructors

`~sparse_matrix() noexcept`

The destructor is responsible for decreasing the reference count of matrices created.

2.4 Modifiers

Addition

`auto operator+=(sparse_matrix const& other) -> sparse_matrix&;`

Exception message "matrices must have identical dimensions" is thrown when the two matrices do not have identical dimensions

Subtraction

`auto operator-=(sparse_matrix const& other) -> sparse_matrix&;`

Exception message "matrices must have identical dimensions" is thrown when the two matrices do not have identical dimensions

Multiplication

`auto operator*=(sparse_matrix const& other) -> sparse_matrix&;`

Multiplication of the two matrices.

Exception message "LHS cols() != RHS rows()" is thrown if the LHS number of columns does not match the RHS number of rows.

Equality

`auto operator==(sparse_matrix const& other) const noexcept -> bool;`

True if both matrices have the same dimensions, and all elements are equal.

element()

`auto element(std::size_t i, std::size_t j, I val) -> void;`

Set the element at (i, j) to the value `val`.

Exception message "values are not in bounds" is thrown when *i* or *j* are outside of the bounds of the matrix.

Note: This operation invalidates all iterators.

Transpose

```
auto transpose() -> sparse_matrix&;
```

This function transposes the matrix.

2.5 Accessors

rows()

```
auto rows() const noexcept -> std::size_t;
```

Returns the number of rows in the matrix

cols()

```
auto cols() const noexcept -> std::size_t;
```

Returns the number of columns in the matrix

element()

```
auto element(std::size_t i, std::size_t j) const -> I const&;
```

Returns a reference to the value of the *(i, j)th* element of the sparse matrix, or 0 if the element is sparse.

Exception message "values are not in bounds" is thrown when *i* or *j* are outside of the bounds of the matrix.

2.6 Range Access

```
auto begin() -> iterator;
```

Returns an iterator to the beginning of the linear matrix representation.

```
auto end() -> iterator;
```

Returns an iterator to the end of the linear matrix representation.

```
auto rbegin() -> reverse_iterator;
```

Returns an iterator to the beginning of the reversed linear matrix representation.

```
auto rend() -> reverse_iterator;
```

Returns an iterator to the end of the reversed linear matrix representation.

2.7 Streams

Input

```
friend auto operator>>(std::istream& is, sparse_matrix& sm) -> std::istream&;
```

Populates `sm` with a serialized representation (see below) of a sparse matrix from `is`.

Output

`friend auto operator<<(std::ostream& os, sparse_matrix const& sm) -> std::ostream&;`

Since it is not feasible to output a dense representation for a very large sparse matrix, you will instead output a serialised sparse matrix. An *m* x *n* matrix with *k* non-zero total entries and with *k_0*, *k_1*, ..., *k_t* non-zero entries in rows *i_0*, *i_1*, ..., *i_t*, each at some matrix location (*i*, *j*) and with some value *v* is serialised as:

$$
\begin{aligned}
&(m, n, k)\\
&(i_0, j_0, v_0)\text{␣}(i_0, j_1, v_1)\text{␣}\cdots\text{␣}(i_0, j_{k_0-1}, v_{k_0-1})\\
&(i_1, j_{k_0}, v_{k_0})\text{␣}(i_1, j_{k_0+1}, v_{k_0+1})\text{␣}\cdots\text{␣}(i_1, j_{k0+k1-1}, v_{k0+k1-1})\\
&\cdots\\
&(i_t, j_{k-k_t}, v_{k-k_t})\text{␣}(i_t, j_{k-k_t+1}, v_{k-k_t+1})\text{␣}\cdots\text{␣}(i_t, j_{k-1}, v_{k-1})
\end{aligned}
$$

All values are parentheses-enclosed, integer tuples, separated by single spaces as shown. Rows should appear in order and on separate lines. Values are in column-index order. No leading or trailing whitespace should be output on each line. A newline should not be output for the last output line. It is C++ convention that the output operator performs minimal formatting and does not emit a trailing newline.

The sample matrix:

$$
\mathbf{B} = \begin{bmatrix}
1 & 0 & 0 & 2 & 0 & 0 & 3 \\
4 & 5 & 0 & 0 & 0 & 0 & 0 \\
0 & 6 & 7 & 0 & 0 & 8 & 0 \\
9 & 0 & 0 & 10 & 11 & 12 & 0 \\
0 & 13 & 0 & 0 & 14 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 15
\end{bmatrix}
$$

Would output the following:

```
(8,7,15)
(0,0,1) (0,3,2) (0,6,3)
(1,0,4) (1,1,5)
(2,1,6) (2,2,7) (2,5,8)
(3,0,9) (3,3,10) (3,4,11) (3,5,12)
(4,1,13) (4,4,14)
(7,6,15)
```

The output stream operator prints nothing if an attempt is made to call a matrix with a 0-dimension (e.g. after move)

2.8 Iterators

`sparse_matrix` is required to have random-access iterators. Below is a subset of the required member functions for such an iterator.

An iterator moves over each non-zero element in left-right, top-bottom order (just as you would read a book in English).

Construction

`iterator(unspecified)`

Constructs an iterator to a specific element in the graph

Dereference

`auto operator*() const -> ranges::common_tuple<std::size_t, std::size_t, const I&>;`

Returns the current *(x, y, value)* tuple *this is pointing to.

`auto operator[](int n) -> ranges::common_tuple<std::size_t, std::size_t, const I&>`

Equivalent to `*(i + n)` where `i` is an iterator.

Traversal

`auto operator++() -> iterator&;` (1)
`auto operator--() -> iterator&;` (2)
`auto operator+=(int n) -> iterator&` (3)
`auto operator-=(int n) -> iterator&` (4)

Moves *this to:
(1): the next adjacent element
(2): the previous adjacent element
(3): the next adjacent element *n* places ahead

(4): the previous adjacent element *n* place before

`auto operator+(int n) const -> iterator;` (5)
`auto operator-(int n) const -> iterator;` (6)

Returns a new iterator to:
(5): the next adjacent element *n* places ahead
(6): the previous adjacent element *n* places before

Comparison

```
auto operator==(iterator const& other) const noexcept -> bool;
```

Returns `true` if *this and other are pointing to elements in the same range, and `false` otherwise.

```
auto operator<(iterator const& other) const noexcept -> bool;
```

Return `true` if *this is ordered linearly before other and `false` otherwise.

Not All Members Have Been Listed

It will be up to you reference the appropriate documentation of Cppreference to fill in the remaining member functions.

2.10 Static Members

`sparse_matrix` is required to have the below static members.

Identity Matrix

```
static auto identity(std::size_t sz) -> sparse_matrix;
```

This static member produces a new identity matrix. The return value is the identity matrix of the sz x sz square matrix.

Exception message "number of dimensions must be greater than zero" thrown if requested identity does not have a strictly positive dimension.

2.11 Further Friends

Transposed matrix

```
friend auto transpose(sparse_matrix const& sm) -> sparse_matrix;
```

This function produces a new matrix that is the transposed matrix of the matrix passed in.

2.12 Exception Handling

When errors are specified to be thrown in the specification, you must throw an error of type matrix_error (defined in the .hpp file). This exception *must* inherit `std::exception`.

3. Performance Requirements

The astute and resourceful (=lazy) reader will quickly realise that, once `element()` is correctly coded, most matrix operations can be implemented as they would be for a dense matrix. This will produce a painfully slow, if correct, solution that is unlikely to pass any of the tests.

The whole idea behind a sparse matrix representation is performance, which cannot be achieved if the operations do not take advantage of this sparse representation. For example, dense matrix multiplication runs in *O(n^3)*. So for two 10^9 x 10^9 matrices, that's 10^27 operations... way too much.

This is why you must implement this as a sparse matrix as described.

4. Usage

To test your code you can write your own tests in `test/q2` (like the one we've provided), or you can test your code manually by building and running `q2-example-usage` we've provided too.