Due Date
The due date is **Friday, Feb. 7, 2003 by 11:59:59 pm**. See the syllabus for late policies and academic integrity policies. See below for submission guidelines.

Sparse Matrices
A matrix is a rectangular (two-dimensional) array of numbers with a series of associated algebraic operations. Matrices are used in a huge number of mathematical operations, although most people first learn about them in solving systems of linear equations. The dimension of a matrix is written as $m \times n$, with $m$ representing the number of rows and $n$ representing the number of columns. In linear algebra, matrix entries are indexed from 1 to $m$ and 1 to $n$, but for simplicity here we will index them from 0 to $m - 1$ and 0 to $n - 1$.

In many cases, the values of $m$ and $n$ are huge, sometimes as large as 1,000,000, but the vast majority of the matrix entries are 0. In fact, often there are only $O(m)$ non-zero entries. In this case, explicitly representing the entries that are 0 is extremely expensive, both in terms of space and time. Therefore, special-purpose “sparse matrix” representations are often developed. You will develop and implement one type of sparse matrix representation in this project.

Before discussing the sparse matrix representation, we will review matrix addition and matrix multiplication. To do this, we need some descriptive notation. Let $A$ be an $m \times n$ matrix. We say that $A_{i,j}$ is the matrix entry in the $i^{th}$ row and the $j^{th}$ column. For example, suppose

$$A = \begin{pmatrix}
5 & 0 & 1 \\
0 & 2 & 0 \\
3 & 4 & 4 \\
0 & 0 & 0 \\
1 & 0 & 3
\end{pmatrix}.$$  

Then, $A_{0,0} = 5$ and $A_{4,2} = 3$ (remember, the indices, which are also called subscripts, start at 0). For the purpose of discussion, let $B$ and $C$ be two more example matrices: let $B$ be $5 \times 3$, with

$$B = \begin{pmatrix}
0 & 0 & 0 \\
1 & 1 & 0 \\
2 & 6 & 0 \\
0 & 3 & 4 \\
1 & 1 & 0
\end{pmatrix},$$
and let $C$ be $3 \times 2$, with

$$C = \begin{pmatrix} 0 & 3 \\ 2 & 3 \\ 0 & 1 \end{pmatrix}.$$  

The two main algebraic operations on matrices that we will define are addition and multiplication. First consider addition. Addition is applied only to pairs of matrices that have the exact same number of rows and the exact same number of columns. Following our example, we are allowed to add $A$ and $B$, but we can’t add either to $C$. Addition occurs by adding the corresponding entries in each matrix. More formally, if $R = S + T$, then

$$R_{i,j} = S_{i,j} + T_{i,j}.$$  

For our example

$$A + B = \begin{pmatrix} 5 & 0 & 1 \\ 1 & 3 & 0 \\ 0 & 3 & 4 \\ 2 & 1 & 3 \end{pmatrix}.$$  

Multiplication is a little more complicated. If $S$ is $m \times n$ and $T$ is $p \times q$, then we can compute $R = S \times T$ if and only if $n = p$. For our ongoing example, we are allowed to multiply $A \times C$ and $B \times C$, but no other combination. In particular, we can’t compute $C \times A$, which means matrix multiplication is not commutative. The formula for matrix multiplication is as follows. With $S$ being $m \times n$ and $T$ being $n \times q$ (since $n = p$), we have

$$R_{i,j} = \sum_{k=0}^{n-1} S_{i,k}T_{k,j},$$  

and $R$ is $m \times q$. For our example,

$$A \times C = \begin{pmatrix} 0 & 16 \\ 4 & 6 \\ 8 & 25 \\ 0 & 0 \\ 0 & 6 \end{pmatrix}.$$  

If matrix multiplication is unfamiliar to you, play with examples until you have the idea fixed.

**Sparse Matrix Representation**

The idea of a sparse matrix representation is to represent only those matrix entries (locations) that have non-zero values. An entry not represented automatically has a value of 0. The non-zero entries must be represented in such a way as to allow efficient access to any given row or any given column of the
matrix. This is crucial for making matrix multiplication fast. Thus, our sparse matrix representation will have a linked list for each row and a linked list for each column of the array. Each non-zero entry will have to be represented only once, but will be entered in two different linked lists, one for its row and one for its column. This is a variation on the idea of a doubly-linked list.

Overall, the representation will have an array (or vector) of pointers to linked lists of rows and an array (or vector) of pointers to linked lists of columns. Here’s a diagram of how this might look for array A, which isn’t very sparse but is small enough for a good demonstration.

In this picture, rows and cols are arrays of pointers and each non-zero entry is depicted with two pointers. Null pointers are indicated with an X.
Your Job

Your job is to implement a templated sparse matrix class based on the above description. This includes implementing constructors, copy constructors, several utility functions, and several important operators, including `+`, `*=`, `=`, and `==`. If the array is \( m \times n \), the computation cost of these operations must not depend on the size of the array, \( mn \), but rather on \( m \) and \( n \) individually (for initialization) and on the number of non-zero entries. (You may assume there are \( O(1) \) entries in any row or column.) For example, in multiplying two \( n \times n \) arrays, \( S \) and \( T \), the cost of calculating the \( i,j \) entry in the result should depend on the number of non-zero entries in row \( i \) of \( S \) and column \( j \) of \( T \). By contrast, with a non-sparse representation, the cost of calculating this entry based on the initial description of multiplication is proportional to \( n \).

Here’s a header file containing prototypes for the class member functions you must implement. You MUST NOT change these function prototypes, but you may add to the class definition:

```cpp
#ifndef sparse_matrix_h_
#define sparse_matrix_h_

template <class T>
class sparse_matrix {

public:
    sparse_matrix( int nrows, int ncols );
    sparse_matrix( const sparse_matrix<T>& old );
    sparse_matrix();

    const sparse_matrix<T>& operator=( const sparse_matrix<T>& old );
    void put( int r, int c, const T& value );
    void make_zero( int r, int c );
    void resize( int nrows, int ncols );
    bool is_zero( int r, int c ) const;
    T get( int r, int c ) const;

    sparse_matrix<T> operator+( const sparse_matrix& rhs ) const;
    sparse_matrix<T> operator*( const sparse_matrix& rhs ) const;
    bool operator==( const sparse_matrix& rhs ) const;

    int num_rows() const;
    int num_cols() const;

    void print_row( ostream& ostr, int r ) const;
    void print_col( ostream& ostr, int c ) const;
};
#endif
```

Your job is to implement the member functions of the class. In doing so,
you will need to specify an internal representation and perhaps several utility functions. Most of these should be self-explanatory, but here are more details on several of them:

- **put** acts to store the value in the array. If there is already a non-zero entry at \( r, c \), this value is changed. If not, this entry must be added. Don’t worry about \( r \) and \( c \) being out of subscript range.

- **get** should return the value stored if it is there, and 0 otherwise.

- **make_zero** should remove an entry from the sparse array if it is there and do nothing otherwise.

- **resize** should have the effect of creating a matrix with all zeroes.

- **print_row** and **print_col** should print only the non-zero entries in the specified row or column.

Many of these are illustrated using the sample main program and sample output posted on the course web page. The output format of your program MUST match the sample output.

Put your implementation at the end of the file `sparse_matrix.h`. This will be the only code you submit. A sample main program that uses the class is provided. You should do careful testing on your own, however. We will be writing an additional main program that tests your code more thoroughly based on much larger matrices and will use this as the basis for grading.

**Grading Criteria**

You will be graded on compilation, program structure, and correct execution of the class member functions. The breakdown will be 30% compilation, 30% structure and 40% correctness. If your program does not compile you will have trouble earning more than 30 points on the project!! All programs must compile using VC++ .NET or g++ (version 2.9.5 or higher, check using `g++ --version`). The structure of your code will be judged for quality of the comments, quality of the data structure design, and especially the logic of the implementation. The comments need not be extremely long; just explain clearly the overall design and then explain the major steps in each function.

**Submission Guidelines**

Your submission must only include two files: the `sparse_matrix.h` file and a brief `readme.txt` file. This file should summarize how correctly you believe your program works and specify which of the two legal compilers you used. (VC++ .NET will be taken as the default.) Here’s how to submit:

1. Copy your files to RCS (RPI’s unix system).
2. Create a tar archive containing the files. The name of the archive file should be your RCS user ID.

3. Copy the tar archive to the correct (!) submission directory.

Here are more details:

**Copying your files to RCS:** There are many ways to copy files from your local machine to RCS (contact the help desk if you do not know how). If you use FTP to copy your files, you must make sure you transfer the files in text format. If you transfer the files in binary format, your submission will be corrupted and you can expect a VERY poor grade.

**Creating a tar archive:** To create a tar archive, use the tar command with three flags c (create), v (verbose), and f (file), followed by the name of the tar file and followed by a list of all the files you want to include (just two for now, but more later). For example, a student with RCS ID `userid` would type

```
tar cvf userid.tar readme.txt sparse_matrix.h
```

**Copying to the submission directory:** The submission directory is `/dept/cs/dsa/project1/`

In the project directory are subdirectories for each section. Copy your tar file to the proper sub-directory. For example if you are in section 3, copy your tar file to `/dept/cs/dsa/project1/section3/`. For user `userid`, this would be done using

```
cp userid.tar /dept/cs/dsa/project1/section3/.
```

You'll find that you can not over-write your submission file after it has been copied. To update your submission use the following numbering system:

- First submission: `userid.tar`
- Second submission: `userid-2.tar`
- Third submission: `userid-3.tar`
- ...

Only your last submission will be graded. The time stamp on the last tar file (check using the command `ls -l`) will determine the submission time of your project!
Notes and Suggestions

- The assignment operator and copy constructor are crucial. When your '+' and '*' operators return a sparse matrix they will call the copy constructor. You can’t prevent it. Spend a lot of time to be sure that you have the copy constructor done correctly.

- Use the put function as much as possible.

- Do not use the get function in your matrix copy constructor, assignment operator, addition operator, or multiplication operator.

- Develop your code incrementally and keep a back-up copy. Don’t get caught making a last minute change to your only copy that messes up the entire program!

- Start early: compiler and execution errors can’t be fixed in a predictable amount of time.

- Start early!

Final Warning

Proper submission is entirely **your responsibility**. Contact your TA if you have any doubts whatsoever about your submission. Do **NOT** submit your project via email. Be sure to keep copies of your files, including your tar file, and **do not change them after submitting**. Don’t even copy them. It is a good idea to keep your submission files, including your tar file, in a protected directory on RCS to have a valid timestamp. After grades are posted, you have exactly one week to resolve all problems. After that week is up, all grades are final.

A project that does not follow the submission guidelines will receive a **10 point deduction**.