CS4115 Week07 Lab Exercise

Lab Objective: In this lab we will refine the `matz` program we began working on last time. We will remove the requirement that the number of rows / columns of the matrix need to be given on the command line, letting the program figure this out for itself as it reads the input. We will then add an option that ignores very small values of the matrix allowing them to be replaced by 0. Here's the lab summary:

❶ Modify `matz` so that it no longer needs to be told how many rows or columns are in the input

❷ Introduce a `-e` (for epsilon, ϵ) option to `matz` so that entries smaller than ϵ are treated as 0

❸ Submit your `matz` program for marking using `handin`

In Detail

❶ When using the input operator, `>>`, to read numbers C++ is very accommodating – usually. When reading from an input stream it works hard to find the next sensible looking number from the input. If necessary, it will skip over non-digits and "white space" including the end-of-line character.

What this means is that when your `matmult` program read its input it was **irrelevant** whether it all came on one line or on three lines or on 18,000 lines with blanks in between. You controlled the number of values read in through your two nested `for`-loops with loop limits already known from the command-line options for `rowct` and `colct`. Wouldn't it be nice if we didn't always have to give this information but could let the program figure it out for itself based on the input it gets?

The same issue arises with our `formatz` compressed matrix specification from last time. We firstly wrote out the number of rows that were to follow and then, for each row, we started it with `k`, the number of non-zero entries on that row of the matrix. The sole reason for that was so that when reading in a matrix stored in the `formatz` format we could set up `for`-loops with the appropriate limits on them. But if we stuck to the convention that **every line of the file corresponds to exactly one row of the matrix** then we could just grab a full line from the file and find what indices for that row were non-zero.

The next generation of the `formatz` specification is shown over. What we are discussing here will certainly be relevant when you go to **read** a file saved in this format but it is also more immediately relevant since I am no longer telling you (via the command-line) how

2.4	0	5.6	0
0	11.5	1.3	5.4
5.6	1.3	0	0
0	5.4	0	0

```
1 2.4 3 5.6
2 11.5 3 1.3 4 5.4
1 5.6 2 1.3
2 5.4
```
formatz

Figure 1: Sample input and output according to the revised `formatz` specification.

many rows or columns there are in the input. Your only knowledge is that there is an exact correspondence between one row of the matrix and one line of the input file.

You should expect the input to `matz` to be that given in the panel on the left in Figure 1 and your output should match that on the right.

The `matz` program should now centre on a single `while`-loop that reads each line of input and processes it. How do we achieve this? The following two panels contain two very common C++ programming idioms. The one on the left loops through the input reading string after string after string. It relies on the stream input operator, `>>`, continuing to return `true` to the `while()` test as long as it has found a string in the input to read in. Once `operator>>` runs out of input `false` gets returned and control falls out of the loop.

```
string str;
while (cin>> str) {
   // do something with
   //   this string
}
```
string by string

```
string line;
while (getline(cin, line)) {
   // do something with
   //   this line
}
```
line by line

The loop in the right hand panel operates in a similar way: the function `getline()` takes as arguments an input stream and a string to fill and returns either `true` (the stream is still flowing) or `false` (no more input). It reads an *entire* line of input up to the end of line marker into the variable `line`. Both of these idioms as written are safe and avoid breaches of security due to buffer overflow that often arise when trying to read an unknown amount of data from some input source in a C++ or `C` program.

This is a valuable trick for you to know: you can now read a line of input a *line* at a time. But now you have to break down that line into its components. In the case at hand the line will be made up of a row of the input matrix, some zero, some not. (In a later lab you will be asked to write a program to read as *input* what you are creating here and the issue will be the same.)

So, how to peel off the `doubles` that the line of input contains? C++ provides a lovely solution to this problem. It allows us to think of the line of input as being a **stream** of input in its own right. There's a little setting up to be done but the essential idea is that we can have another `while` loop that iterates over a given string. C++ calls this a "string stream"

or `istringstream` in the input stream case (there's an output string stream object, too). You will need to add

```
#include <sstream>
```

at the beginning of your program if you want to use this. (Those of you who know a bit about Java will be reminded a bit of the `StringTokenizer` class here.)

The panel below outlines the program structure. There are two nested `while`-loops: the outer one reads a line of input from the file, and the inner one breaks that down into the individual `double` values. You will have to keep track of row index and column index as necessary.

```
// get next full line of text; NB: as text
string line;
while (getline(cin, line))
{
  :
  std::istringstream lstream(line) ;
  double val;

  // peel off values in this line, one at a time
  while (lstream>> val)
  {
    // check if val is 0, etc.
  }
}
```

Core program structure

❷ In addition to "compressing" a matrix the need often arises for treating *very small* values in a matrix as 0. We will specify what we mean by very small on the command line by means of `-e xxx`. As you read values from input anything less than or equal to `xxx` (in absolute value terms) should be treated as 0. Note that this is not a *compulsory* option so `matz` could be run in either of the two ways:

```
matz < m1.out > m1.matz
matz -e 0.0001 < m1.out > m1.matz
```

The file is called `m1.out` because it is the *output* of Week03's program.

So the first thing you will have to do in your program is to check the number of command-line arguments and if more than 1 (the program name is always the first command-line argument and is stored in `argv[0]`) then you have to set up `epsilon`. Here's what I did. You will need to modify your program from last time to compare everything against this value of ϵ.

An ϵ set-up

```
double epsilon;
  :
if (argc > 1 &&
    string(argv[1]) == "-e")
  epsilon = fabs(strtod(argv[2], 0));
```

If there are additional options on the command-line *and* the first one is `-e` then we have work to do. We create an instance of a C++ `string` using whatever is in `argv[1]` and then use the string equality test operator to check if it is equal to the string `-e`. For all you anoraks out there the string equality test operator has signature `string::operator==(string& rightHandSide);` The `string` to the left of the `==` operator (`argv[1]` in our example above) calls the member function `operator==` using the `string` on the right hand side as the argument. The `strtod()` function takes the second command-line argument and converts it to something of type `double`. This will work with either `0.0001` or `1e-5` or `-1e-7`. In case a negative number was given we use `fabs()` to get the absolute value of it.

A sample executable, `matz`, that conforms to this week's specifications is available in this week's class lab directory `~cs4115/labs/week07`.

Here's what I got when I ran

```
matz -e 0.25 < ../week03/m1.in
```

Note how blank lines get written back out.

```
2 0.646261 5 -0.602863 6 0.325489
2 -0.445926 4 -0.556 5 -0.52464 6 0.414701
1 0.358842 3 -0.715326 5 0.412051 6 0.426857
1 -0.461536 2 0.275689 3 0.292509 4 -0.458522 5 0.428733 6 0.481003
1 -0.541986 2 -0.550149 4 0.521111 6 0.354118
1 0.539187 3 0.607526 4 0.38629 6 0.427912



3 0.358842 4 -0.461536 5 -0.541986 6 0.539187
1 0.646261 2 -0.445926 4 0.275689 5 -0.550149
3 -0.715326 4 0.292509 6 0.607526
2 -0.556 4 -0.458522 5 0.521111 6 0.38629
1 -0.602863 2 -0.52464 3 0.412051 4 0.428733
1 0.325489 2 0.414701 3 0.426857 4 0.481003 5 0.354118 6 0.427912
```

❸ You should submit for inspection and marking by us your `matz.cc` file.

The deadline for submitting Week07's assignment without penalty is, as usual, one week from the start of the Weekly lab cycle. Submissions will be accepted until three days after but with a lateness penalty of 10% per day.

Labs that are to be handed in are open for handin at 09.00 on Friday of the week it is *assigned* and, in order to get full marks, are due by 15.00 on Friday of the *following* week; a lateness penalty applies to submissions made until 18.00, Monday after that. This gives you one week to work on each lab that is to be assessed and handin without penalty. The lab sheet will be available for reading earlier in the week so that you can read it beforehand and come to the lab with questions.

Subject to last-minute changes, the planned schedule of lab assignments due for handing in are:

Lab. Week	Assessed	DueDate
Week02	✗	
Week03	✓	Fri, Week04
Week04	✗	
Week05	✗	
Week06	✓	Fri, Week07
Week07	✓	Fri, Week08
Week08	✗	
Week09	✓	Fri, Week10
Week10	✗	
Week11	✓	Fri, Week12