Important message on plagiarism

The single most important point for you to realize before the beginning of your studies at ShanghaiTech is the meaning of “plagiarism”:

Plagiarism is the practice of taking someone else's work or ideas and passing them off as one's own. It is the misrepresentation of the work of another as your own. It is academic theft; a serious infraction of a University honor code, and the latter is your responsibility to uphold. Instances of plagiarism or any other cheating will be reported to the university leadership, and will have serious consequences. Avoiding any form of plagiarism is in your own interest. If you plagiarize and it is unveiled at a later stage only, it will not only reflect badly on the university, but also on your image/career opportunities.

Plagiarism is academic misconduct, and we take it very serious at ShanghaiTech. In the past we have had lots of problems related to plagiarism especially with newly arriving students, so it is important to get this right upfront:

You may…
• … discuss with your peers about course material.
• … discuss generally about the programming language, some features, or abstract lines of code. As long as it is not directly related to any homework, but formulated in a general, abstract way, such discussion is acceptable.
• … share test cases with each other.
• … help each other with setting up the development environment etc.

You may not …
• … read, possess, copy or submit the solution code of anyone else (including people outside this course or university)!
• … receive direct help from someone else (i.e. a direct communication of some lines of code, no matter if it is visual, verbal, or written)!
• … give direct help to someone else. Helping one of your peers by letting him read your code or communicating even just part of the solution in written or in verbal form will have equal consequences.
• … gain access to another one’s account, no matter if with or without permission.
• … give your account access to another student. It is your responsibility to keep your account safe, always log out, and choose a safe password. Do not just share access to your computer with other students without prior lock--out and disabling of automatic login functionality. Do not just leave your computer on without a lock even if it is just for the sake of a 5--minute break.
• … work in teams. You may meet to discuss generally about the material, but any work on the homework is to be done individually and in privacy. Remember, you may not allow anyone to even just read your source code.

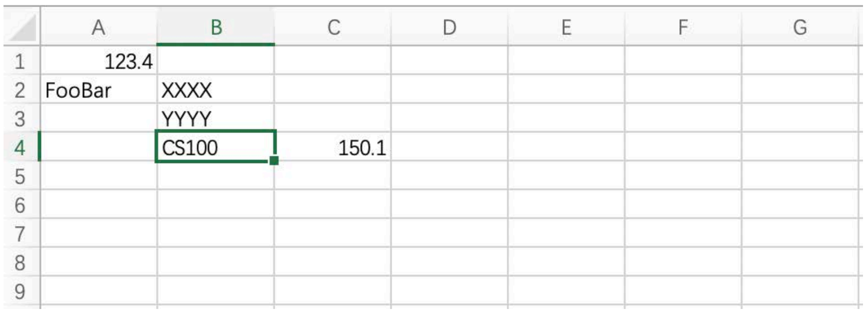
With the Internet, "paste", and "share" are easy operations. Don’t think that it is easy to hide and that we will not find you, we have just as easy to use, fully automatic and intelligent tools that will identify any potential cases of plagiarism. And do not think that being the original author will make any difference. Sharing an original solution with others is just as unethical as using someone else’s work.

CS100 Homework 6 (Spring, 2021)

Submission deadline:

2021-05-24   23:59

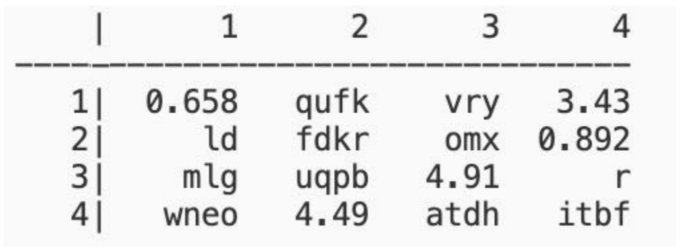
We will implement a simple spreadsheet in this homework, which looks like the Microsoft Excel. The following figure shows the basic structure of a spreadsheet. The first row is called the Column Index (A, B, C, D, E, … in the figure), and the first column is called the Row Index (1, 2, 3, 4, … in the figure). Each cell in this sheet can be identified by a pair of Row Index and Column Index (The circled cell in this figure is called B4 because its row index is 4 and column index is B). Each cell in a spreadsheet can either be a number, a string or Empty. For example, Cell A2, B2, B3 and B4 in following sheet are string cells, Cell A1 and C4 are number cells, and the others are Empty cells.



	A	B	C	D	E	F	G
1	123.4						
2	FooBar	XXXX					
3		YYYY					
4		CS100	150.1				
5							
6							
7							
8							
9							

As you can imagine, Microsoft Excel is a very large project. But don't worry, you are only required implement a very simple spreadsheet in this homework – no GUIs, and only basic functions (print the formatted sheet, sort the sheet by column, hide/display rows and columns).

Also, our spreadsheet uses `int` (starts from 1) for both row and column indexes, here is an example:



```
    |      1      2      3      4
--------------------------------
   1|  0.658   qufk    vry   3.43
   2|     ld   fdkr    omx  0.892
   3|    mlg   uqpb   4.91      r
   4|   wneo   4.49   atdh   itbf
```

Problem 1. Spreadsheet Cells

The basic structure of our spreadsheet is that, each cell is a `SpreadSheetCell` object, and the `SpreadSheet` class holds a 2-dimensional container of pointers to such cells. In this sub-problem, we will implement SpreadSheetCell class and its derived classes. The definition of the base class is:

```
class SpreadSheetCell
{
public:
    SpreadSheetCell();
    ~SpreadSheetCell();

    friend std::ostream& operator<<(std::ostream& os, const SpreadSheetCell& cell);
    virtual bool operator<(const SpreadSheetCell& that) const;
    CellType GetType() const;

protected:
    CellType m_type;
};
```

This class should be an abstract class (i.e., it cannot be instantiated). It should at least have these member functions:

- `GetType()`, returns the type of this cell. The return value is either `CellType::String` or `CellType::Number`
- `operator<`, the operator used to compare this cell with another cell. The comparison rules will be specified below.

Also, in order to conveniently print the content of each cell to `std::cout` (or any `std::ostream`, like `std::ofstream` to print to a file), it is needed to overload the `operator<<` of `std::ostream`. This has been declared for you:

```
  friend std::ostream& operator<<(std::ostream& os, const SpreadSheetCell& cell);
```

Note that, although this function is declared within `class SpreadSheetCell`, <u>it is not a member function of the class</u>. (This means if you define the function outside of your class, specifying `SpreadSheetCell::` for it will be an error.

Since this function is not a member of `class SpreadSheetCell`, the keyword `friend` lets it access private or protected members of your class.

As mentioned before, a non-empty cell either contains strings or numbers (we use double to store numbers in this homework). So, you need to implement two derived classes called `StringSpreadSheetCell` and `NumberSpreadSheetCell` which inherits from the base class. Please follow these instructions when implementing the derived classes:

- String cells can only be constructed with `string`, and Number cells can only be constructed with `double`.
- The `GetType()` function of `StringSpreadSheetCell` will return `CellType::String`, and `NumberSpreadSheetCell` will return `CellType::Number`.
- Printing contents (when `operator<<` is called)
  - For string cells, just print the content.
  - For number cells, you should print the content number with a precision of 3 (use `std::setprecision(3)` when printing it).

    * It's guaranteed that the value in any number cell is between -1000 and 1000.
  - You should not print anything except the content (i.e., No additional '\n' or spaces).
- Comparison (when `operator<` is called)
  - Comparison between number cells will just compare the content value.
  - Comparison between string cells will compare strings in lexicographical order.
  - Number cells are always less than string cells.

*As the parameter of this `operator<` is a `const SpreadSheetCell&`, what you may want to do could be first judge its type by `GetType()`, then create a pointer to it and cast to a const pointer of a derived class[1]. For example:

```
if (that.GetType() == CellType::Number)
{
    const NumberSpreadSheetCell* ptr = (const NumberSpreadSheetCell*)(&that);
    // Then use ptr as a pointer to a NumberSpreadSheetCell.
}
```

This test function may help you to realize how it works (the comment shows the expected output and the reason).

```
void test()
{
    std::vector<SpreadSheetCell*> cells;
    cells.push_back(new NumberSpreadSheetCell(1.5333));
    cells.push_back(new NumberSpreadSheetCell(2.0));
    cells.push_back(new StringSpreadSheetCell("abc"));
    cells.push_back(new StringSpreadSheetCell("bbb"));
    std::cout << std::boolalpha << (*(cells[0]) < *(cells[1])) << std::endl;
```

[1] A better manner that C++ provides is to use `dynamic_cast<>`.

```
    // true (1.5333 < 2.0)
    std::cout << std::boolalpha << (*(cells[1]) < *(cells[0])) << std::endl;
    // false (2.0 > 1.5333)
    std::cout << std::boolalpha << (*(cells[2]) < *(cells[3])) << std::endl;
     // true ("abc" < "bbb")

    std::cout << std::boolalpha << (*(cells[3]) < *(cells[2])) << std::endl;
    // false ("bbb" > "abc")
    std::cout << std::boolalpha << (*(cells[0]) < *(cells[3])) << std::endl;
    // true (numbers < strings)
    std::cout << std::boolalpha << (*(cells[3]) < *(cells[0])) << std::endl;
    // false (strings > numbers)
    std::cout << (*cells[0]) << std::endl;
    // 1.53 (precision = 3)
    std::cout << (*cells[2]) << std::endl;
    // abc (just print it)
    std::cout << cells[0]->GetType() << std::endl;
    // 0 (CellType::Number)
    std::cout << cells[2]->GetType() << std::endl;
    // 1 (CellType::String)
}
```

Submission Guidelines:

When submitting, make sure you have finished the base class `SpreadSheetCell` and its two derived classes. Your code should contain declaration and implementation of these three classes and all related functions. If you also submit code for `SpreadSheet`, make sure that it compiles.

Problem 2. Basic operations of Spreadsheet

Now that we have finished two types of cells, let's make a spreadsheet to hold these cells. When testing problem 1, a `std::vector<SpreadSheetCell*>` is used. Similarly, the spreadsheet will store cells in a 2-dimentional `vector`, and will allow more operations.

```
class SpreadSheet
{
public:
    SpreadSheet(int rows, int cols);
    ~SpreadSheet();

    void UpdateCell(int row, int col, SpreadSheetCell* cell);

    void HideRow(int row);
    void HideColumn(int col);
    void ShowRow(int row);
    void ShowColumn(int col);

    void ExportSheet(std::ostream& os);
private:
    std::vector<std::vector<SpreadSheetCell*> > m_cells;
};
```

In the following specifications, you may see the C++ keyword `nullptr`, abbreviated for null pointer. The keyword `nullptr` defines a null pointer of any type. Much of the cases when using `nullptr` are equivalent to using `NULL` in C.

- The constructor should initialize `m_cells` by rows and cols, and fill it with `nullptr` (or equivalently `NULL`).
- The destructor should delete all cells in `m_cells`, as they are created by `new`.
- The `UpdateCell(int row, int col, SpreadSheetCell* cell)` function puts the given `cell` at position `[row][col]`. If there is already a cell, this function should also delete the original cell to prevent memory leak. If `[row][col]` is off boundary, this does nothing.
- The `Hide` and `Show` functions hide or show a given column or row. A hidden column or row does not show in `ExportSheet`. Its column index or row index will also be hidden.
- Finally, the `ExportSheet(std::ostream& os)` will print the spreadsheet into `os` (you may assume `std::cout`), in the following format:
    - For each row that contains data, there is first a row index of width `DEFAULT_ROW_IDX_WIDTH = 4` (you can use `std::setw`), then a `'|'`. Then print each cell in that row with width `DEFAULT_COLUMN_WIDTH = 7`. If a cell is `nullptr`, you should keep the columns aligned.

- The first two rows of your output do not contain data from your spreadsheet. They are a row of column indices, and a row of separating line. There is no row index for the first row, and the separating line is a full line of `'-'`.
- Every row should end with a `'\n'` or `std::endl`. Do not print any additional endlines.

This is an example test function:

```
void test()
{
    SpreadSheet sheet(4, 4);
    sheet.UpdateCell(1, 1, new NumberSpreadSheetCell(1.1));
    sheet.UpdateCell(1, 3, new NumberSpreadSheetCell(3.4));
    sheet.UpdateCell(2, 4, new StringSpreadSheetCell("foo"));
    sheet.UpdateCell(3, 1, new StringSpreadSheetCell("bar"));
    sheet.UpdateCell(3, 2, new StringSpreadSheetCell("baz"));
    sheet.UpdateCell(4, 2, new NumberSpreadSheetCell(0));
    sheet.HideColumn(3);
    sheet.HideRow(2);
    sheet.ExportSheet(std::cout);
}
```

The desired output should be:

```
    |       1       2       4
-------------------------
   1|     1.1
   3|     bar     baz
   4|               0
```

Submission Guidelines:

Your submission should contain both code for problem 1 and code for the class `SpreadSheet`, with implementations of all related functions except `SortByColumn(int col)`. If your code includes this function, make sure that it compiles.

Problem 3. Sorting and Functor

In this problem, we will add to the SpreadSheet a function to sort the cells by a column. We will be using `std::sort` from `<algorithm>`.

The `std::sort` function could sort an STL container. It needs two parameters, the `begin()` and `end()` of your STL container. You can also provide an optional third parameter, a comparison function of 2 parameters. For example, if you want to sort a `vector<int> vec` in descending order, and suppose `MoreThan(int x, int y)` is a (static) function. You can call `std::sort` by:

```
                std::sort(vec.begin(), vec.end(), MoreThan);
```

Here we want to sort each row of the spreadsheet, according to the content of a given column. See this example that calls `SortByColumn(1);` :

```
Before sort
   |      1      2      3      4      5
----------------------------------------
  1|    1.1           3.4
  2|                         foo
  3|    bar    baz
  4|    0.5      0
  5|    raz
```

```
After sort
   |      1      2      3      4      5
----------------------------------------
  1|    0.5      0
  2|    1.1           3.4
  3|    bar    baz
  4|    raz
  5|                         foo
```

Therefore, we need a comparison function that compares two rows of cells (`std::vector<SpreadSheetCell*>`) by contents of cells at given column. but this function cannot take only 2 parameters. The sorting column should also be passed into this function. A functor could solve this problem.

A functor, or a function object, is a class with overloaded `operator()`. This means we can use this class like calling a function. It’s also allowed to define member variables, because it is also a class.
(For details, refer to https://en.wikipedia.org/wiki/Function_object)

In this way, we can define a functor class named `ColumnComparator`, save the sorting column as its member variable, and define its `operator()` to take only two parameters. Then `std::sort` could be called with a functor as the third parameter.

For this problem, you need to finish the `class ColumnComparator`, and implement the function `void SpreadSheet::SortByColumn(int col);`

As you can see from the example above, the sorting order is ascending, and `NumberSpreadSheetCell < StringSpreadSheetCell < nullptr`. Also, if a `nullptr` is compared against a `nullptr`, let’s define that left side is always less than right side. (i.e. your comparison function should `return true` in this case.) You have already finished comparison between cells, so only cases with `nullptr` needs to be specifically considered.

Submission Guidelines:

Your submission should contain complete code for problem 1, 2, and 3.