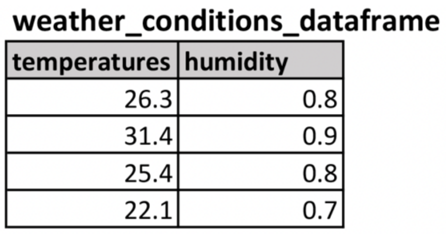
Challenge #2

DataFrames are basic data-structures used heavily in the scope of data-intensive computing. A DataFrame is essentially a 2-dimensional labeled data structure with columns of the same type, and it is designed to support the production and operation with data containers, e.g., datasets. You can think of it as a spreadsheet, but common examples also reflect a table, or even a Python DataFrame (with the constraint to have column data of the same type!). The nice property of a DataFrame is that you can access its columns' contents directly by name. For example, Figure 1 below shows a DataFrame constructed by two columns which are storing temperatures and humidity for a weather forecasting application which produced this data.



weather_conditions_dataframe

temperatures	humidity
26.3	0.8
31.4	0.9
25.4	0.8
22.1	0.7

Figure 1: A weather application DataFrame.

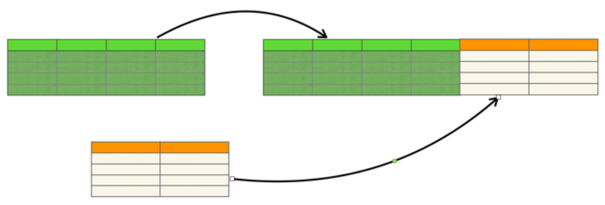
We have already provided both the declaration of the DataFrame class (storing elements of type double), and basic methods to deal with DataFrames themselves, for example constructors, setters, and getters.

Your task is to implement two methods, "**hstack**" and "**join**", declared as follows:

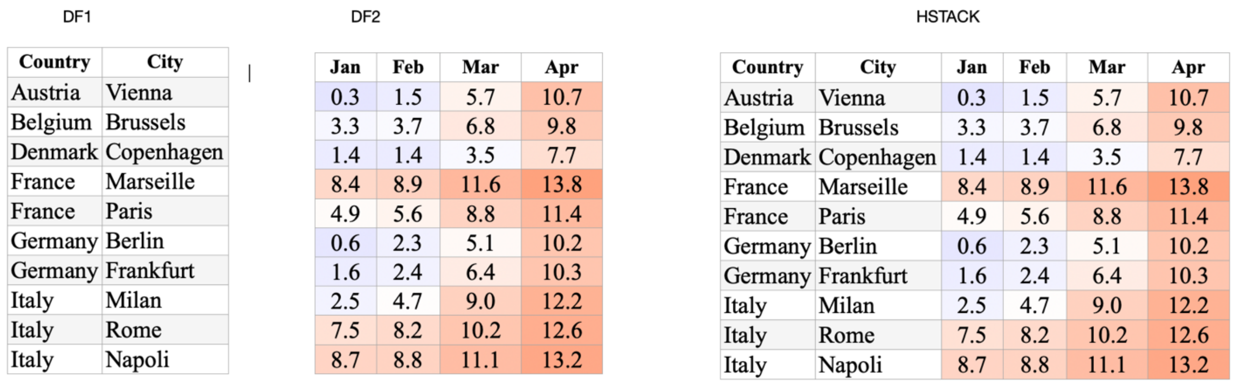
```
DataFrame hstack(DataFrame &otherDataFrame);

DataFrame join(DataFrame &otherDataFrame,

               string onMyCol,

               string onColOfOther);
```

Moreover, you will have to implement consistently ANY other method you eventually need to perform such operations.

First, the **hstack** method will "H"orizontally pack the two DataFrames similar to the equivalent Python function. The expected behavior shall, for example, result in the condition represented on the figure below:



A more concrete example of the hstack task behavior is reported below:



DF1

Country	City
Austria	Vienna
Belgium	Brussels
Denmark	Copenhagen
France	Marseille
France	Paris
Germany	Berlin
Germany	Frankfurt
Italy	Milan
Italy	Rome
Italy	Napoli

DF2

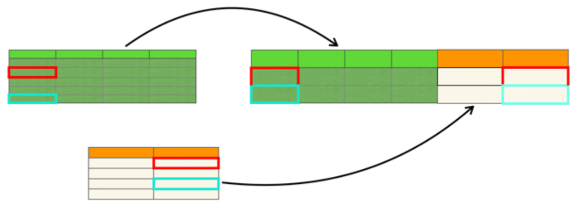
Jan	Feb	Mar	Apr
0.3	1.5	5.7	10.7
3.3	3.7	6.8	9.8
1.4	1.4	3.5	7.7
8.4	8.9	11.6	13.8
4.9	5.6	8.8	11.4
0.6	2.3	5.1	10.2
1.6	2.4	6.4	10.3
2.5	4.7	9.0	12.2
7.5	8.2	10.2	12.6
8.7	8.8	11.1	13.2

HSTACK

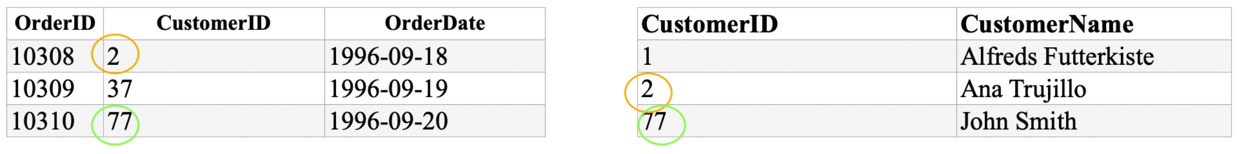
Country	City	Jan	Feb	Mar	Apr
Austria	Vienna	0.3	1.5	5.7	10.7
Belgium	Brussels	3.3	3.7	6.8	9.8
Denmark	Copenhagen	1.4	1.4	3.5	7.7
France	Marseille	8.4	8.9	11.6	13.8
France	Paris	4.9	5.6	8.8	11.4
Germany	Berlin	0.6	2.3	5.1	10.2
Germany	Frankfurt	1.6	2.4	6.4	10.3
Italy	Milan	2.5	4.7	9.0	12.2
Italy	Rome	7.5	8.2	10.2	12.6
Italy	Napoli	8.7	8.8	11.1	13.2

(data from https://en.wikipedia.org/wiki/List_of_cities_by_average_temperature)

The second task at hand is to implement the "join" operator; **join** works as the same usual command in SQL, namely by executing a Cartesian product on tables and filtering rows out based on a criterion (logic expression) between values in different columns.

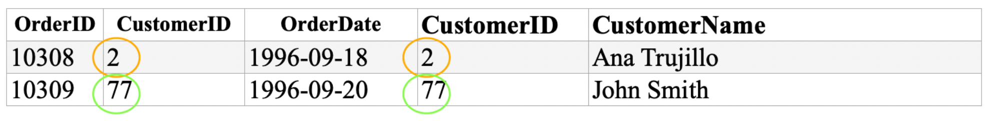


An operational example is reported below:



OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

CustomerID	CustomerName
1	Alfreds Futterkiste
2	Ana Trujillo
77	John Smith



OrderID	CustomerID	OrderDate	CustomerID	CustomerName
10308	2	1996-09-18	2	Ana Trujillo
10309	77	1996-09-20	77	John Smith

Here the intent is to join tables based on **CustomerID: we find rows with the same CustomerID, i.e. row 1 on left dataset AND row 2 on right (having same id ==2 on both sides), and row 3 on left dataset and row 3 on right (having id == 77 on both sides). Note that colors in the previous figure provide a correspondence between the starting tables (top) and the resulting concrete ones (bottom).**

More generally, suppose you have two DataFrames, df1 and df2 represented below:

df1

A	B	C
-2	(0)	4
1	3	(0)

df2

D	E	F
6	7	8
1	10	(0)

As per its specification, **hstack** returns:

df1_A	df1_B	df1_C	df2_D	df2_E	df2_F
-2	(0)	4	6	7	8
1	3	0	1	10	(0)

<u>To avoid any issue with columns with the same name, your code must create a new DataFrame with every column from df1 and every column form df2 adding also a prefix to colum name.</u>

While **join** -supposing that you call **d1.join(d2, “A”, “D”)**, (i.e., same value “1” in both specified column “A” and “D”))- will return:

df1_A	df1_B	df1_C	df2_D	df2_E	df2_F
1	3	(0)	1	10	(0)

Your code must scan every row on df1 and consider the values in column A.
In df1 we have:

- -2 that identifies NO row on df2 (we don’t have any -2 in any cell of column df2_D)
- 1 that identifies the 2nd row on df2, the row: 1,10, 0

So the code will align the values 1,3,0 from df1 with the values 1,10,0 from df2.

Please read files cases_output1.txt, cases_output2.txt and cases_output3.txt where you can see some examples of output you should obtain from different DataFrames using the “include”s in main file:

```
#include "TestValues1.h"
//#include "TestValues2.h"
//#include "TestValues3.h"
```

As a final note, **you have to optimize the data structure choice to manage SPARSE data, i.e., many values will be zeros. However, you can assume that in the join operations the columns involved in the equality test include only non zero elements (this is why zeros are in**

parentheses in the examples above, you should not store their values!!!). Moreover, you have to optimize the worst case complexity in your access operations.

Submission details

As in previous challenges, pay VERY MUCH attention to adding/removing comments and commented text/code as requested and as appropriate.

As in previous challenge, we expect you will make various tests using different values, e.g., as per the following:

```
#include "TestValues.h"
```

To participate to this challenge do the following steps:

1) Unzip the CLion project and run as a test
2) **Modify only the code where you find "YOUR CODE HERE"**
3) Change the values inside the "TestValues.h" file to test your code: we will also test your implementation by changing this file with other values.
4) **Don't change the matrix class name or method signatures from what is already assigned.**
5) Test with some values
6) When done, cleanup our code, re-zip your code and submit to WeBeep.
7) Zip file must generate a FLAT folder (i.e. without subfolders) containing ONLY original cpp and h files (other files and folders will be skipped).
8) As in previous challenge, the ZIP file must be EXACTLY named as:
   codicepersona.ZIP, so for example 10047232.zip
   Different names will be rejected.

Note that in the evaluation of this second challange we will be very strict: if your code will not work because you have not followed the indications above, you did not organise the files properly and/or you did not name your zip file as specified, YOUR SUBMISSION WILL NOT BE EVALUATED.