Chair for Clinical Bioinformatics
M. Sc. Georges Schmartz
M. Sc. Pascal Hirsch
B. Sc. Matthias Flotho
B. Sc. Saurabh Pandey

Programming Course 2nd Assignment

Hand in until 08.12.20 at 11:59 pm

General remarks

Please make sure that you follow all rules stated in the general remarks PDF in the moodle. The number of barbells 🏋 for each task describes the expected difficulty. The maximum of number of barbells per task will be 4.
Note that beginning with this assignment sheet, tutors and automatic tests will assess error-handling and bounds checking mechanisms of your implementations by providing invalid input.

Task 1 (8 P.): 🏋

1. What is the difference between a `struct` and a `class`? (0.5 P.)
2. Why should the public interface to a class be as small as possible? (1 P.)
3. What is the difference between `const` and `static`? (0.5 P.)
4. Why should you distinguish between functions that modify objects and those that do not? (1 P.)
5. What is the concept and effect of `inline` functions? When would you prefer `inline` functions over function calls? (0.5 P.)
6. What is a destructor and when is it called? What is the typical consequence of its usage? (1 P.)
7. In which context are virtual functions used? (1 P.)
8. What is the semantic result of combining the keywords `friend` and `virtual` for a class function? Explain and state reasons for your observations. (1 P.)
9. When would you make a virtual member function pure? (0.5 P.)
10. What are pitfalls of multiple inheritance? (0.5 P.)
11. What is the difference between a concrete class and an interface? (0.5 P.)

Task 2 (10 P.): ↖↘

Create a class `SimpleNode` that stores its content in a `double`. Additionally, a `SimpleNode` stores its successor node in a `unique_ptr<SimpleNode>`, allowing to create a linked list. The following operations need to be implemented:

1. `SimpleNode(content)`, default initializer for the head.
2. Copy constructor and copy assignment operator, making a deep copy.
3. Move constructor and move assignment operator.
4. Member function `insert(successor_node)`, which adds the given `successor_node` as successor to the called node. The `successor_node` should keep any successor nodes it has. If the called node already has a successor, insert the `successor_node` and all its possible successors in between the called node and its current successor.
5. Member function `next`, returning the successor node as `SimpleNode`, returning `NULL` if there is none.
6. Member function `is_tail`, returning whether there is no successor node.
7. Member function `length`, providing the number of nodes, counting itself and all following successor nodes.
8. Operator() to access and/or modify the stored content.

Task 3 (10 P.): ↖↘↖↘

For this task you should compute gene clusters in which each gene is at most at distance d of at least one other gene of the cluster and all being located on the same chromosome. To compute the distance between two genes compare their middle positions.
The genes will be passed as tab-separated text file as first argument. The allowed distance will be passed as second argument and the output file as third argument.
The input file consists of three columns: `chromosome start stop`, start and stop represent a closed interval and begin at one. The output file consists of four columns, i.e. the same as the input file and one additional column containing the cluster index. To assign each gene to a cluster, add a new column to the input file: `cluster=i`, with i being the cluster index (starting at 1). The clusters should be sorted in ascending coordinate order, i.e., with `chr1` coming before `chr2` (and before `chrX` and `chrY`), then according to their starting position. The order of the genes in the input file can be assumed to be in sorted in coordinate order. If this is not the case your program should exit with an appropriate error message and error code 1.

Sample command:

```
./task3 sample_input.txt 2000 sample_output.txt
```

Sample input:

```
chr1 100 21000
chr1 250 18000
chr1 21000 24000
chr2 1300 10800
```

Sample output:

```
chr1 100 21000 cluster=1
chr1 250 18000 cluster=1
chr1 21000 24000 cluster=2
chr2 1300 10800 cluster=3
```

Task 4 (17 (10+7) P.): (↘↘ + ↘↘↘)

Write a program that takes two strings as arguments and outputs the best alignment score on the standard output. To obtain all the points, output after the best score the best global alignment(s). If there are multiple best global alignments report them all, in any order, and separate them by an empty line (trailing empty lines are allowed). Use the Needleman-Wunsch algorithm to implement it. The scores are defined as follows: match = 1, mismatch = -1 and gap = -1. Sample input:
Sample input:

```
GCATGCT
GATTACA
```

Sample output:

```
0
GCATG-CT
G-ATTACA

GCA-TGCT
G-ATTACA

GCAT-GCT
G-ATTACA
```

Task 5 (25 P.): ↘↘↘

Implement a class `Dataframe` that can manage a dynamic number of columns (of identical length), where each column has its own value type. Internally, the data should be stored in a `std::list<vector<ColType>>`, named `data`. Make sure that `ColType` is capable of encapsulating any valid C++ type, as dataframes are typically

used to represent tables of heterogeneous data types. For example, if the first column is of type `int`, only integers can be stored in that column, while a second column of type `std::string` can only store strings along the rows. In addition, a dataframe can identify rows and columns not only by their position (aka index) but also through an optional set of names for both rows and cols (duplicates are not allowed). To add value beyond the simple storage of heterogeneous data, your `Dataframe` class must implement the following member-functions and operations:

1. `Dataframe()`, default initialized.
2. Copy constructor and copy assignment operator (generation by the compiler is allowed).
3. Move constructor and move assignment operator (generation by the compiler is allowed).
4. Member functions `nrows` and `ncols` providing the number of rows and columns.
5. Member function `get<T>(i,j)` to access and/or modify the element of type T stored at row i and column j. The data access is 0-indexed. This should return a reference to the original element (ensure const-correctness!).
6. Member function `get<T>(r,c)` to access and/or modify the element stored at row named r and column named c. This should return a reference to the original element (ensure const-correctness!).
7. Member functions `set_colnames` and `set_rownames` to set the column names and row names.
8. Member functions `clear_colnames` and `clear_rownames` to remove column and row names.
9. Member functions `has_colnames` and `has_rownames` to check if column names or row names have been specified.
10. Member function `add_column<T>(vector<T>)` that adds a new column with entries of type `T`.
11. Member function `remove_column` that removes a column either by index or by name.
12. Member function `swap_columns` that allows to exchange the position of two columns, both either being given by numeric index parameters or by column name.
13. Member function `apply` that executes on a given column identified by either index or name the passed lambda function on each row element of the column (functions that modify the elements are allowed).

14. Member function `order_by` that allows to sort the rows (in ascending order by using `operator <`) using a given column identified by either index or name. This has the effect that all other columns will be resorted according to the newly defined row ordering.

15. Operator `<<` to output data. The data should be tab separated for the columns and each row is in a new line. If column names are given they shall be outputted in the first line. In case row names are present they should be outputted as first element of each line and the first line must start with a tab.

16. Operator `==` and `!=` that check the elements of the dataframes for (in)equality. When comparing columns of different types the equality operator should always return false.

Throw an exception in case of any invalid usage of the implemented functions. Empty `Dataframe`s are generally allowed and should be constructible.