Exercise 9: Singly linked list template

Deadline: 28th May 2021

A *singly linked list* is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail). Each element in a linked list is called a *node*. A single node contains data and a pointer to the next node which helps in maintaining the structure of the list. The first node is called the *head*; it points to the first node of the list and helps us access every other element in the list. The last node, also sometimes called the *tail*, points to NULL which helps us in determining when the list ends.

Task

Define an class template `list<T>` for a singly linked list. The definition of the list should be written in accordance with the art of programming dynamic data structures - define a fully functional list node `node<T>` as a private class nested in the `list<T>` wrapper class, having a user-friendly interface with dictionary operations (inserting an element into a given position, deleting an element with a given value, searching for element, counting all items, etc).

```
template<typename T>
class list {
    class node {
    // ...
    };
private:
    node *head;
public:
// ...
};
```

Place the class template definition in the `calc` namespace.

List objects should be copyable (copy constructor, copy assignments, move constructor, and move assignment). Complete the list definition with initialization through `initializer_list<T>`. Don't forget the operator << to write the whole list to the output stream.

Next, define two function templates for working with lists: `issorted()` to check if the list is ordered and `sort()` to order all element in the list. Place these definitions in the `calc` namespace too. The templates should have two parameters: the type of data stored in the list and the trait implementing the comparison of two elements. Trait should be the default parameter in the templates, set to a function object that implements comparison by operator <. Also define another trait that implements the comparison by operator >. Make sure that you specialize traits for pointers, in particular for the `const char*`.

Finally, write a program that reliably test your list implementation. The program should interact with the user.