program.h

```
#include "program_menu/program_menu.h"

class program {

    program_menu *program_menu{};
    cultural_place_list *cultural_places;

    void init_program_menu();

public:

    program();

    void run();

};
```

program.cpp

```
#include "program.h"
#include "cultural_place_list.h"
#include "program_menu\commands\add_cult_zav.h"
#include "program_menu\commands\add_rate.h"
#include "program_menu\commands\show_cult_zavs.h"
#include "program_menu\commands\write_to_file.h"
#include "program_menu\commands\read_from_file.h"
#include "program_menu\commands\update_cult_zav.h"
#include "program_menu\commands\delete_cult_zav.h"
#include "program_menu\commands\sort_cult_zav_by_rate.h"
#include "program_menu\commands\exit.h"

program :: program(){

    cultural_places = new class cultural_place_list();
    init_program_menu();

}

void program :: init_program_menu() {

    program_menu = new class program_menu;

    program_menu -> add_command(new class add_cult_zav());
    program_menu -> add_command(new class add_rate());
    program_menu -> add_command(new class show_cult_zavs());
    program_menu -> add_command(new class write_to_file());
    program_menu -> add_command(new class read_from_file());
    program_menu -> add_command(new class update_cult_zav());
    program_menu -> add_command(new class delete_cult_zav());
    program_menu -> add_command(new class
sort_cult_zav_by_rate());
```

```
    program_menu -> add_command(new class exit());

};

void program :: run(){

    program_menu -> run(cultural_places);

}

linked_list.h

#pragma once

#include <algorithm>
#include <iostream>
#include <memory>
#include <functional>

#include "iterator.h"
#include "sortable.h"
#include "cultural_place.h"

template <class T, typename
std::enable_if<std::is_base_of<sortable, T>::value>::type* =
nullptr>
class linked_list
{
    template <class T>
    class linked_list_iterator;

    template <class T>
    class Node
    {
        friend class linked_list_iterator<T>;
        friend class linked_list<T>;

        Node() : next(nullptr), previous(nullptr), data(nullptr)
{}
        explicit Node(const T &data) : data(data),
next(nullptr), previous(nullptr) {}
        Node<T> *next;
        Node<T> *previous;
        T data;
    public:
    };
    template <class T>
    class linked_list_iterator : public abstract_iterator<T>
    {
        Node<T>* p;
    public:
        linked_list_iterator(Node<T>* p) : p(p) {}
        linked_list_iterator(const linked_list_iterator& other)
```

```
: p(other.p) {}
        void operator++(int) override { p = p->next; }
        bool operator==(const linked_list_iterator& other) {
return p == other.p; }
        bool operator!=(const linked_list_iterator& other) {
return p != other.p; }
        explicit operator bool() override { return p; }
        T& operator*() const final { return p->data; }
        linked_list_iterator<Node<T>> operator+(int i)
        {
            linked_list_iterator<TNode> iter = *this;
            while (i-- > 0 && iter.p)
            {
                ++iter;
            }
            return iter;
        }
    };

    typedef Node<T> node;

    std::size_t size;
    node * head;
    node * tail;

    void init()
    {
        size = 0;
        head = nullptr;
        tail = nullptr;
    }

public:
    typedef linked_list_iterator<T> iterator;

    linked_list() { init(); }

    ~linked_list()
    {
        cout << "nothing to worry about" << endl;
    }

    void push_back(const T &value)
    {
        node *n = new node(value);
        if (tail) {
            tail->next = n;
            n->previous = tail;
        } else {
            head = n;
        }
        tail = n;
        size++;
    }

    void push_front(const T &value)
    {
        node *n = new node(value);
        if (head) {
            head->previous = n;
            n->next = head;
        } else {
            tail = n;
        }
        head = n;
        size++;
    }

    void remove(T *item){
        node * current = head;
        while(current){
            if (&(current->data) == item){
                if (current->next) current->previous->next = 
current->next;
                else tail = current->previous;
                if (current->previous) current->next->previous = 
current->previous;
                else head = current->next;
                delete current;
                return;
            }
            current = current->next;
        }
        // нетуту
    }

    void sort(){
        bool is_sorted = false;
        node * current = head;
        while (!is_sorted){
            is_sorted = true;
            while (current && current->next){
                if (current->data.get_sort_value() > current-
>next->data.get_sort_value()){
                    if (current->previous){
                        node * node1 = current->previous;
                        node * node2 = current;
                        node * node3 = current->next;
                        node * node4 = current->next->next;
                        if (node1) {
                            if (node3) node1->next = node3;
                            else tail = node2;
                        }
                        else head = node2;
                        if (node3) node3->next = node2;
                        else tail = node2;
                        node2->next = node4;
                        node4->previous = node2;
                        if (node3) node2->previous = node3;
                        if (node3 && node1) node3->previous =
node1;
                    }
                    is_sorted = false;
                    current = current->next;
                }
            }
        }
    }

    int get_size() const
    {
        return size;
    }

    iterator * begin()
    {
        return new class iterator(head);
    }

    void clear()
    {
        init();
    }
};
```

program_menu.h

```
#include <vector>
#include "command.h"

using namespace std;

class program_menu {

    vector<command*> commands;

public:

    void add_command(command *command);

    void run(cultural_place_list * cultural_places);

};
```

program_menu.cpp

```
#include <windows.h>

#include "program_menu.h"
```

```
void program_menu :: add_command(command *command) {

      commands.push_back(command);
}

void program_menu :: run(cultural_place_list * cultural_places) {
    int i;
    do{
        for (i = 0; i < commands.size(); i++){
            cout << i + 1 << " - " << commands[i]->get_name() << endl;
        }
        cin >> i;

        while (cin.fail() || i < 1 || i > commands.size()){
            cout << "Error input :(" << endl << "Try again:";
            cin.clear();
            rewind(stdin);
            cin >> i;
        }

        try{
            commands[i - 1]->execute(cultural_places);
        }

        catch (exception& except){
            cout << except.what() << endl;
        }

    } while (i != commands.size());
}
```

command.h

```
#ifndef UNTITLED_COMMAND_H
#define UNTITLED_COMMAND_H

#include <iostream>

#include "../cultural_place_list.h"

using namespace std;

class command {
public:

    virtual string get_name() = 0;

    virtual void execute(cultural_place_list * cultural_places)
= 0;
};
```

```
#endif
```

cultural_place_list.h

```cpp
#pragma once

#include "iterator.h"
#include "linked_list.h"
#include "cultural_place.h"

class cultural_place_list{
    linked_list<cultural_place> *cultural_places;
public:

    cultural_place_list();

    abstract_iterator<cultural_place> * get_iterator();

    void add_cultural_place(const cultural_place& cultural_place);

    void delete_cultural_place(cultural_place * cultural_place);

    void sort_cultural_places_by_rate();

    cultural_place& find_by_name(const string& name);
};
```

cultural_place_list.cpp

```cpp
#include "cultural_place_list.h"

cultural_place_list :: cultural_place_list () {
    cultural_places = new class linked_list<cultural_place>();
};

abstract_iterator<cultural_place> * cultural_place_list ::
get_iterator() {
    return cultural_places->begin();
}

void cultural_place_list :: add_cultural_place(const cultural_place&
cultural_place){
    cultural_places->push_back(cultural_place);
}

void cultural_place_list :: delete_cultural_place(cultural_place *
cultural_place){
    cultural_places->remove(cultural_place);
}

void cultural_place_list :: sort_cultural_places_by_rate(){
    cultural_places->sort();
}

cultural_place& cultural_place_list :: find_by_name(const string&
name) {
    abstract_iterator<cultural_place>  * i = get_iterator();

    while (*i){
        if ((**i).name == name){
           return **i;
        }
        (*i)++;
    }
    throw exception ("Cultural place not found :(");
}
```