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

#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

#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 :("**);  
}