**Лабораторная работа №2. Наследование в С++**

1. **Вариант задания №11**

Создать класс Container для какого-либо типа данных с методами push, pop, isEmpty и front. На его основе реализовать классы Stack (стек) и Queue (очередь). Размер контейнера задается при его создании.

1. **Спецификации классов**
   1. **Спецификация класса Container**

В базовом классе Container заданы общие виртуальные функции, которые будут в дальнейшем перегружены в производных классах

* 1. **Спецификация класса Queue**

Является потомком класса Container, хранит в себе объект шаблонного класса List и его размер, а также оператор вывода.

* 1. **Спецификация класса Stack**

Является потомком класса Container, хранит в себе объект шаблонного класса List и его размер, а также оператор вывода.

* 1. **Спецификация класса Item**

В шаблонном классе Item содержится поле данных и 2 указателя на объекты класса Item

* 1. **Спецификация класса List**

Шаблонный класс List реализует двухсвязный циклический список. Содержит поле класса Item и методы добавления и удаления в начало и конец списка, методы, возвращающие начало и конец списка, а также оператор вывода.

1. **Используемые математические алгоритмы и т. д.**

В данной работе классы очереди и стека были реализованы на основе двухсвязного циклического списка. Для реализации методов push и pop были необходимо удаление и добавление в начало и конец списка.

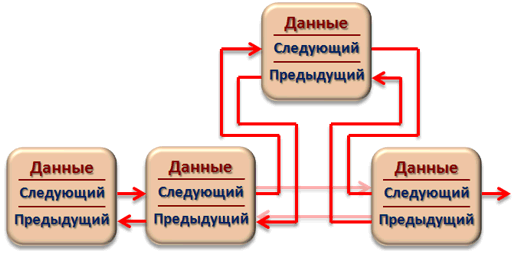


Рисунок - добавление элемента в список

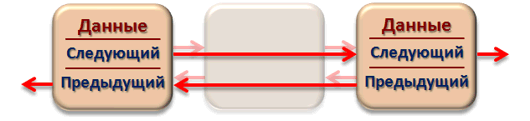


Рисунок - удаление элемента из списка

1. **Диаграмма классов**

**A close up of text on a white background

Description automatically generated**

1. **Листинг программы**

**lab2.cpp**

#include <iostream>

#include "interactor.h"

using namespace std;

int main()

{

Interactor it;

it.init("clients.txt", "food.txt");

return 0;

}

**interactor.h**

#ifndef INTERACTOR\_H

#define INTERACTOR\_H

#include <iostream>

#include <fstream>

#include "queue.h"

#include "stack.h"

#include "food.h"

#include "customer.h"

class Interactor

{

private:

Queue<Customer> \_clients;

Stack<Food> \_food;

void \_help() const;

public:

Interactor();

~Interactor();

void init(const std::string& clients\_list, const std::string& food\_list);

};

void Interactor::\_help() const

{

std::cout <<"===========================================\n";

std::cout << "Disclaimer: \nYou have a restaurant.\n";

std::cout << "He displays his dishes on the window.\n";

std::cout << "There is a line of customers to him.\n";

std::cout <<"Each client wants to buy the freshest dish.\n";

std::cout <<"===========================================\n";

std::cout << "Commands: \n";

std::cout << "1) Show clients\n";

std::cout << "2) Show dishes\n";

std::cout << "3) Customer buy a dish\n";

std::cout << "4) Add new dish\n";

std::cout << "5) Add new client\n";

std::cout << "6) Help\n";

std::cout << "0) Exit\n";

std::cout <<"===========================================\n";

}

Interactor::Interactor(): \_clients(Queue<Customer>()), \_food(Stack<Food>()) {}

Interactor::~Interactor() {}

void Interactor::init(const std::string& clients\_list, const std::string& food\_list)

{

std::ifstream fin(clients\_list);

int num\_of\_clients;

fin >> num\_of\_clients;

for (int i = 0; i < num\_of\_clients; ++i)

{

std::string name;

int money;

fin >> name >> money;

\_clients.push( Customer(name,money) );

}

fin.close();

std::ifstream in(food\_list);

int num\_of\_food;

in >> num\_of\_food;

for (int i = 0; i < num\_of\_food; ++i)

{

std::string name;

int cost;

in >> name >> cost;

\_food.push(Food(name, cost));

}

in.close();

\_help();

int command = 1;

while (command)

{

std::cout << "#Command: ";

std::cin >> command;

std::string name;

unsigned int x;

switch (command)

{

case 1:

std::cout << \_clients;

break;

case 2:

std::cout << \_food;

break;

case 3:

if (\_clients.isEmpty())

{

std::cout << "There are no customers in the queue.\n";

}

else if (\_food.isEmpty())

{

std::cout << "There are no dishes in the stack.\n";

}

else if (\_clients.front().buyFood(\_food.front()))

{

std::cout << \_clients.front().getNameOfClient()

<< " bought " << \_food.front().getNameOfFood()

<< " for $ " << \_food.front().getCostOfFood() << '\n';

\_clients.pop();

\_food.pop();

}

else

{

std::cout << "The client does not have enough money for food.\n";

\_clients.pop();

}

break;

case 4:

std::cout << "Enter the name of the dish and its cost: ";

std::cin >> name >> x;

\_food.push(Food(name, x));

break;

case 5:

std::cout << "Enter the name of the client and the amount of his money: ";

std::cin >> name >> x;

\_clients.push(Customer(name, x));

break;

case 6:

\_help();

break;

default:

break;

}

}

}

#endif

**container.h**

#ifndef CONTAINER\_H

#define CONTAINER\_H

template <class T> class Container

{

public:

Container() {};

~Container() {};

virtual void push(const T& val) = 0;

virtual void pop() = 0 ;

virtual bool isEmpty() const = 0;

virtual T front() const = 0;

virtual int size() const = 0;

};

#endif */\* CONTAINER\_H \*/*

**item.h**

#ifndef ITEM\_H

#define ITEM\_H

template <class T> class Item

{

public:

T data;

Item<T>\* next;

Item<T>\* prev;

Item();

Item(const T& val);

~Item();

};

template <typename T>

Item<T>::Item(): next(this), prev(this) {}

template <typename T>

Item<T>::Item(const T& val): data(val), next(this), prev(this) {}

template <typename T>

Item<T>::~Item(){}

#endif */\* ITEM\_H \*/*

**list.h**

#ifndef LIST\_H

#define LIST\_H

#include "item.h"

#include <stdexcept>

#include <iostream>

template <class T> class List

{

private:

Item<T>\* \_items;

public:

List();

~List();

void addToTail(const T& val);

void addToHead(const T& val);

void deleteFromHead();

void deleteFromTail();

T head() const;

T tail() const;

*// Linker gave an error for the implementation of an operator outside the class*

friend std::ostream& operator<<(std::ostream& out, const List<T>& object)

{

if (object.\_items == nullptr)

{

out << "No elements\n";

return out;

}

Item<T>\* tmp = object.\_items;

int cnt = 1;

do

{

out << cnt << ") " << tmp->data;

tmp = tmp->next;

cnt++;

} while (tmp != object.\_items);

return out;

}

};

template <typename T>

List<T>::List(): \_items(nullptr) {}

template <typename T>

List<T>::~List()

{

while (\_items != nullptr)

{

this->deleteFromHead();

}

}

template <typename T>

void List<T>::addToTail(const T& val)

{

Item<T>\* tmp = new Item<T>(val);

if (\_items == nullptr)

{

\_items = tmp;

return;

}

tmp->prev = \_items->prev;

tmp->next = \_items;

\_items->prev->next = tmp;

\_items->prev = tmp;

return;

}

template <typename T>

void List<T>::addToHead(const T& val)

{

Item<T>\* tmp = new Item<T>(val);

if (\_items == nullptr)

{

\_items = tmp;

return;

}

tmp->prev = \_items->prev;

tmp->next = \_items;

\_items->prev->next = tmp;

\_items->prev = tmp;

\_items = \_items->prev;

}

template <typename T>

void List<T>::deleteFromHead()

{

if (\_items == nullptr)

throw std::logic\_error("Container has no elements");

if (\_items->next == \_items)

{

delete \_items;

\_items = nullptr;

return;

}

Item<T>\* tmp = \_items->next;

\_items->prev->next = \_items->next;

\_items->next->prev = \_items->prev;

delete \_items;

\_items = tmp;

}

template <typename T>

void List<T>::deleteFromTail()

{

if (\_items == nullptr)

throw std::logic\_error("Container has no elements");

if (\_items->next == \_items)

{

delete \_items;

\_items = nullptr;

return;

}

\_items = \_items->prev;

this->deleteFromHead();

}

template <typename T>

T List<T>::head() const

{

if (\_items != nullptr)

{

return \_items->data;

}

else

{

throw std::logic\_error("Container has no elements");

}

}

template <typename T>

T List<T>::tail() const

{

if (\_items != nullptr)

{

return \_items->prev->data;

}

else

{

throw std::logic\_error("Container has no elements");

}

}

#endif */\* LIST\_H \*/*

**queue.h**

#ifndef QUEUE\_H

#define QUEUE\_H

#include "container.h"

#include "list.h"

template <class T> class Queue : public Container<T>

{

private:

int \_size;

List<T> \_que;

public:

Queue();

~Queue();

void push(const T& val);

void pop();

bool isEmpty() const;

int size() const;

T front() const;

friend std::ostream& operator<<(std::ostream& out, const Queue<T>& object)

{

out << "----------------------------\n";

out << "Queue has " << object.\_size << " elements:\n";

out << object.\_que << "\n";

out << "----------------------------\n";

return out;

}

};

template <typename T>

Queue<T>::Queue() : \_size(0), \_que(List<T>()) {}

template <typename T>

Queue<T>::~Queue() {}

template <typename T>

void Queue<T>::push(const T& val)

{

\_que.addToTail(val);

\_size++;

}

template <typename T>

void Queue<T>::pop()

{

if (\_size)

{

\_que.deleteFromHead();

\_size--;

}

else

{

throw std::logic\_error("Queue is already empty");

}

}

template <typename T>

bool Queue<T>::isEmpty() const

{

return (\_size == 0);

}

template <typename T>

T Queue<T>::front() const

{

return \_que.head();

}

template <typename T>

int Queue<T>::size() const

{

return \_size;

}

#endif */\* QUEUE\_H \*/*

**stack.h**

#ifndef STACK\_H

#define STACK\_H

#include "container.h"

#include "list.h"

template <class T> class Stack : public Container<T>

{

private:

List<T> \_stack;

int \_size;

public:

Stack();

~Stack();

void push(const T& val);

void pop();

bool isEmpty() const;

int size() const;

T front() const;

friend std::ostream& operator<<(std::ostream& out, const Stack<T>& object)

{

out << "----------------------------\n";

out << "Stack has " << object.\_size << " elements:\n";

out << object.\_stack << "\n";

out << "----------------------------\n";

return out;

}

};

template <typename T>

Stack<T>::Stack() : \_size(0), \_stack(List<T>()) {}

template <typename T>

Stack<T>::~Stack() {}

template <typename T>

void Stack<T>::push(const T& val)

{

\_stack.addToTail(val);

\_size++;

}

template <typename T>

void Stack<T>::pop()

{

if (\_size)

{

\_stack.deleteFromTail();

\_size--;

}

else

{

throw std::logic\_error("Stack is already empty");

}

}

template <typename T>

bool Stack<T>::isEmpty() const

{

return (\_size == 0);

}

template <typename T>

T Stack<T>::front() const

{

return \_stack.tail();

}

template <typename T>

int Stack<T>::size() const

{

return \_size;

}

#endif */\* STACK\_H \*/*

**food.h**

#ifndef FOOD\_H

#define FOOD\_H

#include <string>

class Food

{

private:

std::string \_name;

unsigned int \_cost;

public:

Food(std::string name, unsigned int cost);

~Food();

std::string getNameOfFood() const;

unsigned int getCostOfFood() const;

friend std::ostream& operator<<(std::ostream& out, const Food& object);

};

Food::Food(std::string name, unsigned int cost): \_name(name), \_cost(cost) {}

Food::~Food() {}

std::string Food::getNameOfFood() const

{

return \_name;

}

unsigned int Food::getCostOfFood() const

{

return \_cost;

}

std::ostream& operator<<(std::ostream& out, const Food& object)

{

out << "The \"" << object.\_name << "\" dish costs " << object.\_cost << "$\n";

return out;

}

#endif *// FOOD\_H*

**customer.h**

#ifndef CLIENT\_H

#define CLIENT\_H

#include <string>

#include "food.h"

class Customer

{

private:

std::string \_name;

unsigned int \_money;

public:

Customer(std::string name, unsigned int money);

~Customer();

std::string getNameOfClient() const;

unsigned int getClientMoney() const;

bool buyFood(const Food& food);

friend std::ostream& operator<<(std::ostream& out, const Customer& object);

};

Customer::Customer(std::string name, unsigned int money): \_name(name), \_money(money) {}

Customer::~Customer() {}

std::string Customer::getNameOfClient() const

{

return \_name;

}

unsigned int Customer::getClientMoney() const

{

return \_money;

}

bool Customer::buyFood(const Food& food)

{

unsigned int cost = food.getCostOfFood();

if (\_money >= cost)

{

\_money -= cost;

return true;

}

else

{

return false;

}

}

std::ostream& operator<<(std::ostream& out, const Customer& object)

{

out << "Customer " << object.\_name << " has " << object.\_money << "$\n";

return out;

}

#endif *// CLIENT\_H*

Автор Преподаватель

Миронченко Павел Дмитриевич \_\_\_\_\_\_\_\_\_(подпись) Оценка\_\_\_\_\_\_

Подпись \_\_\_\_\_\_ Дата \_\_\_\_\_\_\_\_