Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное учреждение высшего образования

«Пермский национальный исследовательский политехнический университет»

ПНИПУ

**Лабораторная работа по ООП**

**«№6»**

Выполнил:

студент группы РИС-23-1б

Кривошеин Александр Антонович

Проверила:

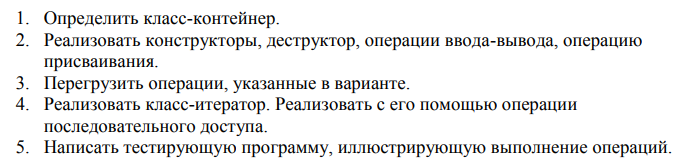
доцент кафедры ИТАС

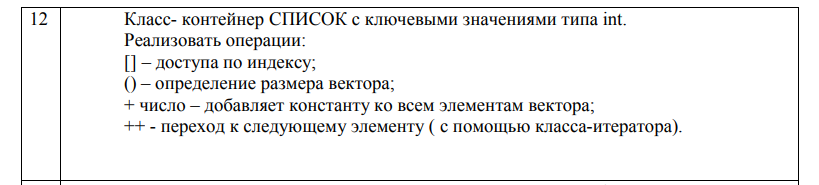
Полякова Ольга Андреевна

2024 г.

**Разработка алгоритма**

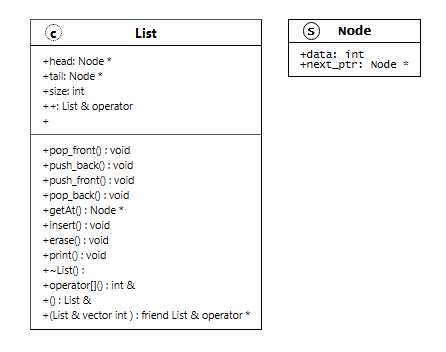
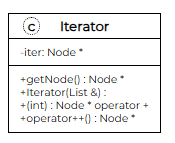
**Постановка задачи:**





**Анализ задачи:**

1. Создадим несколько файлов заголовков и файлов с кодом для удобной работы.
2. Создадим класс-контейнер список List, который будет хранить указатели на головной элемент и конечный элемент списка.
3. Список будет использовать дополнительную структуру Node для реализации узлов списка.
4. Для удобного прохода по списку реализуем итератор Iterator, оперирующий указателями на узлы списка.
5. В главном файле программы создадим список и продемонстрируем все реализованные операции.



Код на C++:

main.cpp:

#include <iostream>

#include <vector>

#include <ctime> // Для srand и time

#include "List.h"

#include "Iterator.h"

using namespace std;

int main() {

    srand(static\_cast<unsigned int>(time(0))); // Инициализация генератора случайных чисел

    List list;

    // Добавление случайных значений в список

    for (int i = 0; i < 5; ++i)

        list.push\_back(rand() % 100);

    list.print();

    // Проверка доступа к элементу

    cout << "Element at index 2: " << list[2] << endl;

    // Изменение размера списка

    list(10);

    // Увеличение каждого элемента на 10

    list += 10;

    list.print();

    return 0; // Явное возвращение 0

}

iterator.h:

#pragma once

#include "List.h"

class Iterator {

private:

    Node \*iter;

public:

    Node \*getNode();

    Iterator(List &a);

    Node \*operator+(int a);

    Node \*operator++();

    Node \*operator++(int);

};

list.h:

#pragma once

#include <iostream>

#include <vector>

#include "Node.h" // Включаем Node.h для структуры Node

class List {

public:

    Node \*head = nullptr;

    Node \*tail = nullptr;

    int size = 0;

    void pop\_front();

    void push\_back(int);

    void push\_front(int);

    void pop\_back();

    Node \*getAt(int);

    void insert(int, int);

    void erase(int);

    void print();

    ~List();

    int &operator[](int);

    List &operator()(int);

    List &operator+=(int);

    friend List &operator\*(List &a, std::vector<int> b);

};

list.cpp:

#include <iostream>

#include <vector>

#include "List.h"

#include "Iterator.h"

using namespace std;

void List::pop\_front() {

    if (head == nullptr)

        return;

    if (head == tail) {

        delete tail;

        head = tail = nullptr;

        return;

    }

    Node \*node = head;

    head = node->next\_ptr;

    delete node;

    --size;

}

void List::push\_back(int data) {

    Node \*node = new Node;

    node->data = data;

    if (head == nullptr)

        head = node;

    if (tail != nullptr)

        tail->next\_ptr = node;

    tail = node;

    ++size;

}

void List::push\_front(int data) {

    Node \*node = new Node;

    node->data = data;

    node->next\_ptr = head;

    head = node;

    if (tail == nullptr)

        tail = node;

    ++size;

}

void List::pop\_back() {

    if (tail == nullptr)

        return;

    if (head == tail) {

        delete tail;

        head = tail = nullptr;

        return;

    }

    Node \*node = head;

    while (node->next\_ptr != tail) {

        node = node->next\_ptr;

    }

    node->next\_ptr = nullptr;

    delete tail;

    tail = node;

    --size;

}

Node \*List::getAt(int k) {

    if (k < 0)

        return nullptr;

    Node \*node = head;

    for (int n = 0; node && n != k; ++n) {

        node = node->next\_ptr;

    }

    return node;

}

void List::insert(int k, int data) {

    Node \*left = getAt(k);

    if (left == nullptr)

        return;

    Node \*right = left->next\_ptr;

    Node \*node = new Node;

    node->data = data;

    left->next\_ptr = node;

    node->next\_ptr = right;

    if (right == nullptr)

        tail = node;

    size++;

}

void List::erase(int k) {

    if (k < 0)

        return;

    if (k == 0) {

        pop\_front();

        return;

    }

    Node \*left = getAt(k - 1);

    Node \*node = left->next\_ptr;

    if (node == nullptr)

        return;

    Node \*right = node->next\_ptr;

    left->next\_ptr = right;

    if (node == tail)

        tail = left;

    delete node;

    --size;

}

void List::print() {

    for (Iterator iterator(\*this); iterator.getNode() != nullptr; iterator++) {

        cout << iterator.getNode()->data << ' ';

    }

    cout << endl << endl;

}

List &List::operator+=(int k) {

    for (Iterator iterator(\*this); iterator.getNode() != nullptr; iterator++) {

        iterator.getNode()->data += k;

    }

    return \*this;

}

List::~List() {

    while (head != nullptr)

        pop\_front();

}

int &List::operator[](int a) {

    Node \*node = getAt(a);

    if (node) {

        return node->data;

    }

    throw out\_of\_range("Index out of range"); // Обработка выхода за границы

}

List &List::operator()(int k) {

    while (size > k) {

        pop\_back();

    }

    while (size < k) {

        push\_back(0);

    }

    return \*this;

}

iterator.cpp:

#include <iostream>

#include "Iterator.h"

using namespace std;

Node \*Iterator::getNode() {

    return this->iter;

}

Iterator::Iterator(List &a) {

    this->iter = a.head;

}

Node \*Iterator::operator+(int a) {

    Node \*temp = this->iter;

    for (int i = 0; i < a && temp; ++i) {

        temp = temp->next\_ptr;

    }

    return temp;

}

Node \*Iterator::operator++() {

    if (iter) {

        iter = iter->next\_ptr;

    }

    return iter;

}

Node \*Iterator::operator++(int) {

    Node \*oldIter = iter;

    if (iter) {

        iter = iter->next\_ptr;

    }

    return oldIter;

}

**node.h**

#pragma once

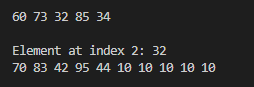
struct Node {

    int data;

    Node \*next\_ptr = nullptr;

};

Пример работы программы:

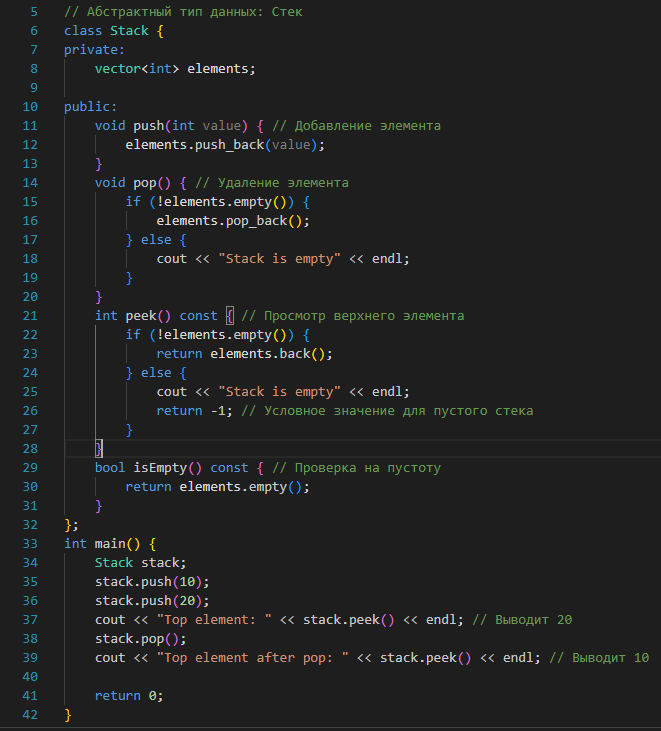


Контрольные вопросы

1. Что такое абстрактный тип данных? Привести примеры

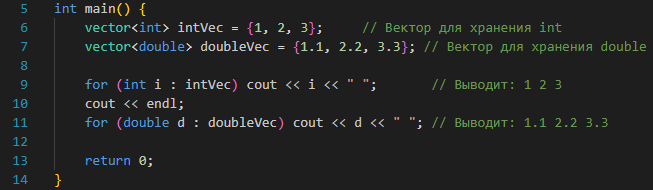
АТД.

АТД - тип данных, определяемый только через операции, которые могут выполняться над соответствующими объектами безотносительно к способу представления этих объектов. Примеры АТД включают строки, списки, деревья, множества, стеки очереди.



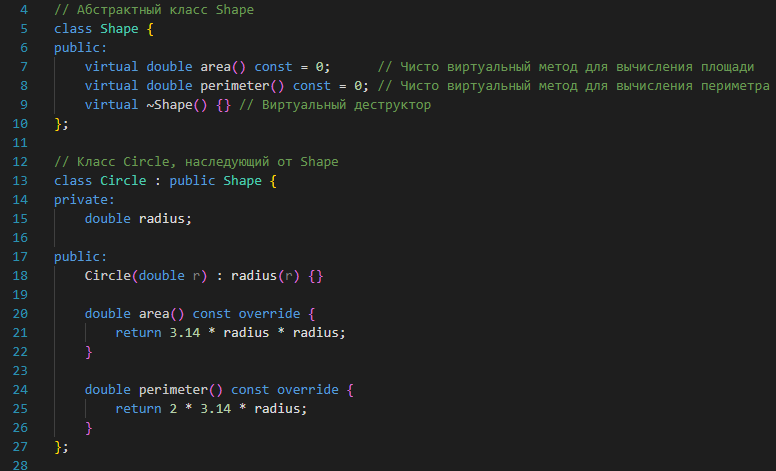
1. Привести примеры абстракции через параметризацию.

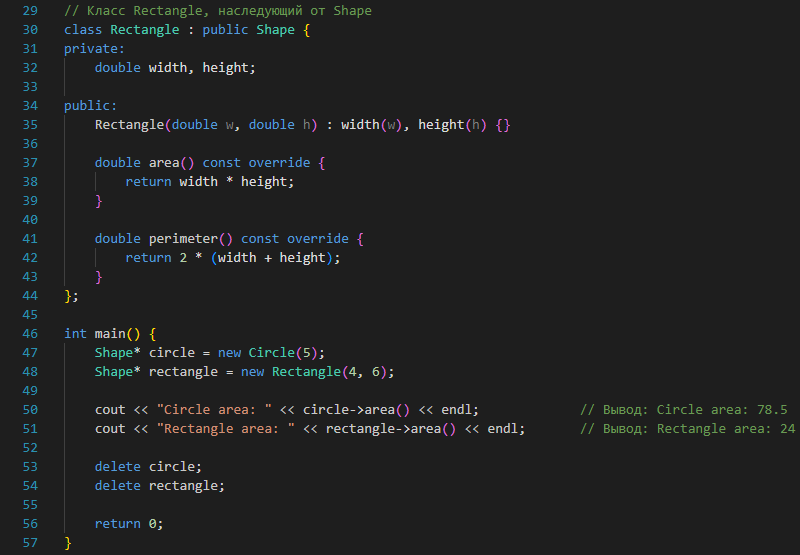
Абстракция через параметризацию позволяет создавать универсальные функции и классы, которые могут работать с любыми типами данных, задаваемыми параметром. Это достигается использованием шаблонов в C++, которые позволяют задавать тип данных в момент вызова функции или создания объекта. Благодаря этому, код становится более гибким и повторно используемым.



1. Привести примеры абстракции через спецификацию

Абстракция через спецификацию позволяет пользователям работать с классами и функциями, не заботясь о деталях их реализации, благодаря определению четкого интерфейса. Это достигается через использование абстрактных классов, интерфейсов и чисто виртуальных функций. Пользователь сосредотачивается на том, что делает класс или функция, а не на том, как это реализовано.





1. Что такое контейнер? Привести примеры.

Контейнер — это структура данных, предназначенная для хранения и управления коллекцией однотипных элементов. Контейнеры обеспечивают удобные методы для добавления, удаления и доступа к элементам, а также могут поддерживать различные операции и алгоритмы, которые упрощают работу с данными



1. Какие группы операций выделяют в контейнерах?

Среди всех операций контейнера можно выделить несколько типовых групп:

* Операции доступа к элементам, которые обеспечивают и операцию замены значений элементов;
* Операции добавления и удаления элементов или групп

элементов;

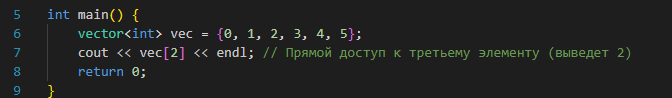
* Операции поиска элементов и групп элементов;
* Операции объединения контейнеров;
* Специальные операции, которые зависят от вида контейнера.

1. Какие виды доступа к элементам контейнера существуют?

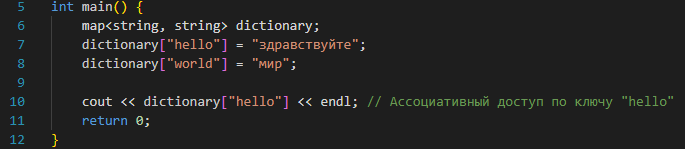
Привести примеры.

Доступ к элементам контейнера бывает: последовательный, прямой и ассоциативный.

Прямой доступ – это доступ по индексу. Например, a[10] – требуется найти элемент контейнера с номером 10.

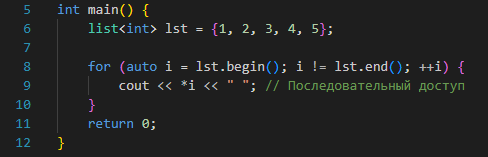


Ассоциативный доступ также выполняется по индексу, но индексом будет являться не номер элемента, а его содержимое. Пусть имеется контейнер –словарь, в котором хранится информация, состоящая, как минимум из двух полей: слово и его перевод. Индексом может служить слово, например, a[“word”].



При последовательном доступе осуществляется перемещение от элемента к элементу контейнера. Набор операций последовательного доступа включает следующие:

* Перейти к первому элементу;
* Перейти к последнему элементу;
* Перейти к следующему элементу;
* Перейти к предыдущему элементу;
* Перейти на n элементов вперед;
* Перейти на n элементов назад;
* Получить текущий элемент.



1. Что такое итератор?

Итератор – это объект, который обеспечивает последовательный доступ к элементам контейнера.

1. Каким образом может быть реализован итератор?

Итератор может быть реализован как часть класса-контейнера в виде набора методов:

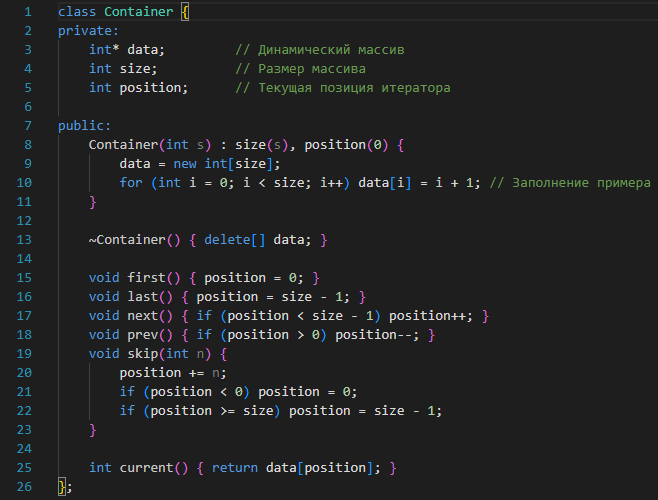
v.first() перейти к первому элементу

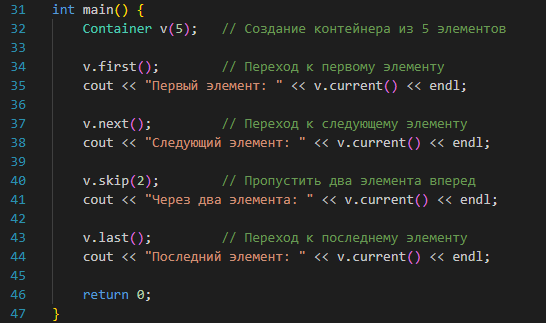
v.last() перейти к последнему элементу

v.next() перейти к следующему элементу

v.prev() перейти к предыдущему элементу v.skip(n) перейти на n элементов вперед v.skip(-n) перейти на n элементов назад

v.current() получить текущий элемент





1. Каким образом можно организовать объединение контейнеров?

* Простое сцепление двух контейнеров: в новый контейнер попадают сначала элементы первого контейнера, потом второго, операция не коммутативна.
* Объединение упорядоченных контейнеров, новый контейнер тоже будет упорядочен, операция коммутативна.
* Объединение контейнеров как объединение множеств, в новый контейнер попадают только те элементы, которые есть хотя бы в одном контейнере, операция коммутативна.
* Объединение контейнеров как пересечение множеств, в новый контейнер попадают только те элементы, которые есть в обоих контейнерах,

операция коммутативна.

* Для контейнеров-множеств может быть еще реализована операция вычитания, в контейнер попадают только те элементы первого контейнера, которых нет во втором, операция не коммутативна.
* Извлечение части элементов из контейнера и создание нового контейнера. Эта операция может быть выполнена с помощью конструктора, а часть контейнера задается двумя итераторами.

1. Какой доступ к элементам предоставляет контейнер,

состоящий из элементов «ключ-значение»?

Ассоциативный доступ.

1. Как называется контейнер, в котором вставка и удаление элементов выполняется на одном конце контейнера?

Стек.

1. Какой из объектов (a, b, c, d) является контейнером?
2. int mas=10;
3. 2. int mas;
4. 3. struct {char name[30]; int age;} mas;
5. 4. int mas[100];

Ответ: d

1. Какой из объектов (a,b,c,d) не является контейнером?
2. int a[]={1,2,3,4,5};
3. 2. int mas[30];
4. 3. struct {char name[30]; int age;} mas[30];
5. 4. int mas;

Ответ: d

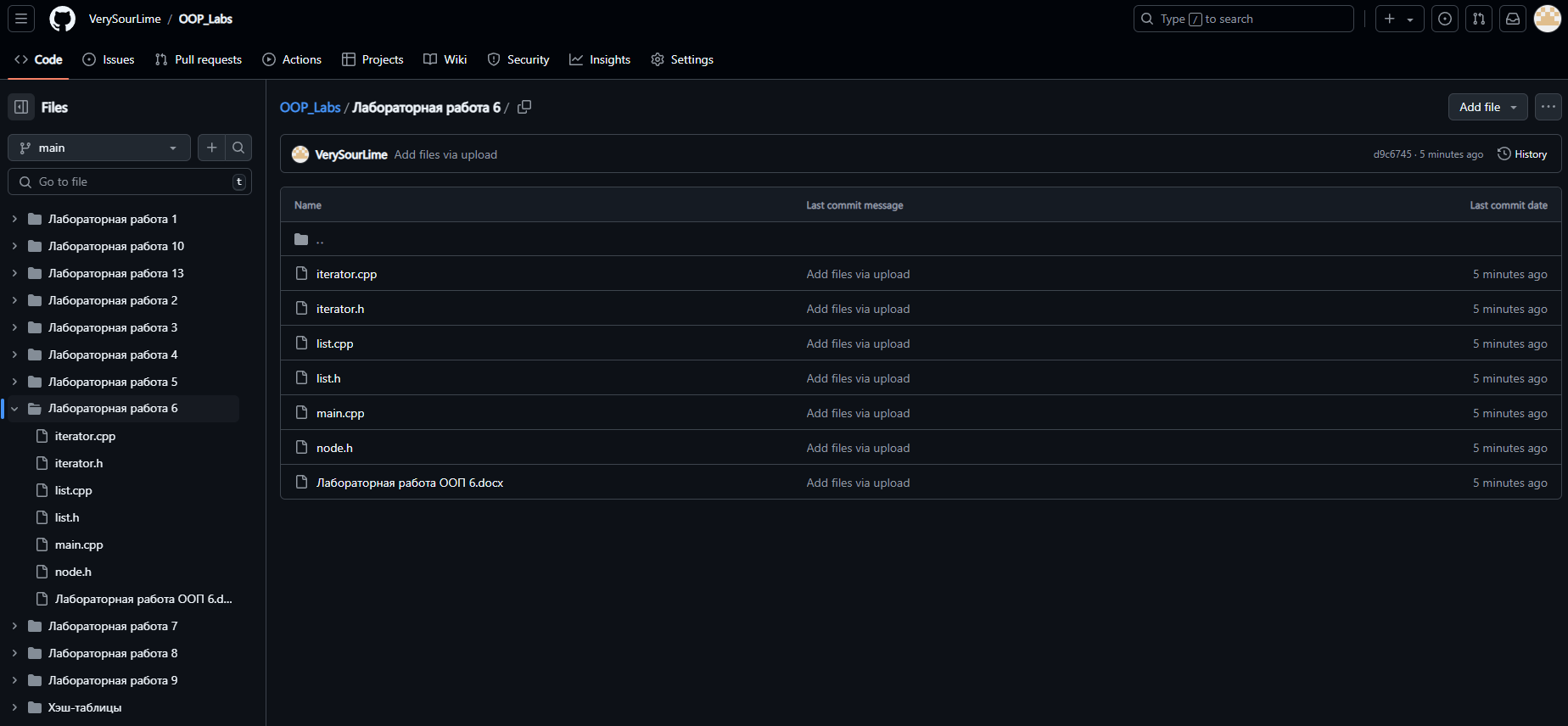
1. Контейнер реализован как динамический массив, в нем определена операция доступ по индексу. Каким будет доступ к элементам контейнера?

Прямой доступ.

1. Контейнер реализован как линейный список. Каким будет доступ к элементам контейнера?

Последовательный доступ.

Скриншот из GitHub:



GitHub: https://github.com/VerySourLime/OOP\_Labs