Министерство образования и науки Российской Федерации

Федеральное государственное образовательное автономное учреждение высшего образования

"Пермский национальный исследовательский политехнический университет"

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №6

Дисциплина: Основы алгоритмизации и программирования

Тема: АТД. Контейнеры.

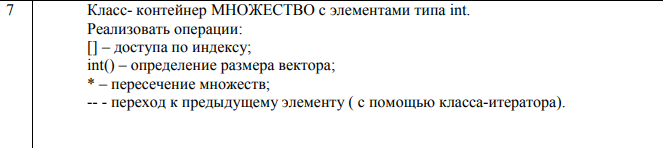
|  |  |
| --- | --- |
| Выполнил работу | |
| Студент группы РИС-22-1б | |
| Бреднев М.П. | |
|  | |
| Проверил работу | |
| Доцент кафедры ИТАС | |
| Полякова О.А. | |
|  | |

Пермь – 2023

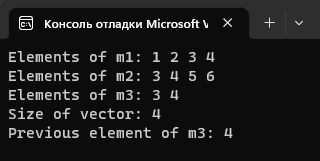
**Анализ предметной области**

**Постановка задачи**

1. Определить класс контейнер.
2. Реализовать конструкторы, деструктор, операции ввода-вывода, операцию присваивания.
3. Реализовать итератор. Реализовать с его помощью операции последовательного доступа.
4. Написать тестирующую программу, иллюстрирующую выполнение операций.



**Тестирование программы**

****

*Рис. 1 – Тестирование программы.*

# Ответы на 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.

template <typename T>

T & Stack<T>::push(){ //операция добавления элементов в АТД типа стек

return head->data;

}

2. Привести примеры абстракции через параметризацию.

template <typename T>

class node {

T data;

node\* next, \*prev;

}

3. Привести примеры абстракции через спецификацию.

class complex {

double re, im;

friend complex operator+(complex, complex);

public:

complex() { re = im = 0;}

complex(double r) { re = r; im = 0; }

complex(double r, double i) { re = r; im = i; }

};

complex operator+(complex a1, complex a2) {

return complex(a1.re + a2.re, a1.im + a2.im);

}

Для поддержки смешанной арифметики комплексных и действительных чисел (Re x, In y) + double A требуется специфицировать перегруженную функцию.

complex operator+(complex,complex);

complex operator+(complex,double);

complex operator+(double,complex);

Абстракция через спецификацию достигается за счет того, что операции

представляются как часть типа (абстракция вычислений, перегрузка операторов).

4. Что такое контейнер? Привести примеры.

Контейнер – это объект. Имя контейнера – это имя переменной. Контейнер, так же как и другие объекты, обладает временем жизни. Время жизни контейнера в общем случае не зависит от времени жизни его элементов. Элементами контейнера могут любые объекты, в том числе, и другие контейнеры.

class queue() {

int size;

node \*head, \*tail;

…

};

//в основной функции

queue\* q; //контейнер-очередь

float a = 7.5;

q->head = a; //головой очереди q является float переменная

queue \*c;

c->tail = q; //хвостом очереди c является контейнер queue

5. Какие группы операций выделяют в контейнерах?

Операции доступа к элементам

template <typename T>

class queue() {

int size;

node \*head, \*tail;

node & getHead() { return head->data; }

…

};

* Операции замены значений элементов;

void queue::setHead(node head){ this->head = head; }

* Операции добавления и удаления элементов или групп элементов;

T queue::pop(){

T data = tail->data;

size–;

tail = tail->prev;

tail->next = nullptr;

return data;

}

* Операции поиска элементов и групп элементов;

int queue::find\_index(T data) {

int counter = 0;

node \* curr = this->head;

while (counter < size) {

if (curr->data = data) return counter;

curr = curr->next;

counter++;

}

}

* Операции объединения контейнеров;

queue queue::merge(queue q1, queue q2) {

q1->tail->next = q2->head;

q1->tail = q2->tail;

delete q2;

return q1;

}

* Специальные операции, которые зависят от вида контейнера.

bool queue::isCircular(queue q) {

return (q->tail->next == q->head); }

6. 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 (с учетом индексации с нуля)

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 будут данные поля, например, q[T data]. Поле, с содержимым которого ассоциируется элемент контейнера, называется ключом или полем доступа. Элемент, с которым ассоциируется ключ, называется

значением. Контейнер, который представляет ассоциативный доступ, состоит из пар «ключ-значение» (ключ - T data, значение - node name).

При последовательном доступе осуществляется перемещение от элемента к элементу контейнера.

Например, дана очередь, и для того, чтобы найти индекс элемента, требуется поэлементно пройти по каждому узлу.

int queue::find\_index(T data) {

int counter = 0;

node \* curr = this->head;

while (counter < size) {

if (curr->data = data) return counter;

curr = curr->next;

counter++;

}

}

7. Что такое итератор?

Итератор (перечислитель) — интерфейс, предоставляющий доступ к элементам контейнера и навигацию по ним. Главное предназначение итераторов заключается в предоставлении возможности пользователю обращаться к любому элементу контейнера при сокрытии внутренней структуры контейнера от пользователя. Это позволяет контейнеру хранить элементы любым способом при допустимости работы пользователя с ним как с простой последовательностью или списком. Итератор можно реализовать как класс, представляющий такой же набор операций.

В С++ итератор реализуется как класс, который имеет такой же интерфейс, как и указатель для совместимости с массивами.

8. Каким образом может быть реализован итератор?

class iterator() {

friend class queue; //поддержка работы с классом очередей

public:

iterator() { elem = nullptr; }

…

iterator & operator++() {  
 elem = elem->next; //реализация части кода с переходом на следующий элемент

}

…

};

9. Каким образом можно организовать объединение контейнеров?

queue queue::merge(queue q1, queue q2) {

q1->tail->next = q2->head;

q1->tail = q2->tail;

delete q2;

return q1;

}

10. Какой доступ к элементам предоставляет контейнер, состоящий из элементов «ключ-значение»?

Ассоциативный доступ (индексом будут данные поля, например, q[T data]. Поле, с содержимым которого ассоциируется элемент контейнера, называется ключом или полем доступа. Элемент, с которым ассоциируется ключ, называется значением)

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

Стек

12. Какой из объектов (a,b,c,d) является контейнером?

a. int mas=10; //переменная типа int

b. 2. int mas; //переменная типа int

c. 3. struct {char name[30]; int age;} mas; //структура

d. 4. int mas[100]; //массив переменных int

Ответ: d

13. Какой из объектов (a,b,c,d) не является контейнером?

a. int a[]={1,2,3,4,5}; //заполненный массив переменных int

b. 2. int mas[30]; //пустой массив переменных int

c. 3. struct {char name[30]; int age;} mas[30]; //массив структур

d. 4. int mas; //переменная типа int

Ответ: d

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

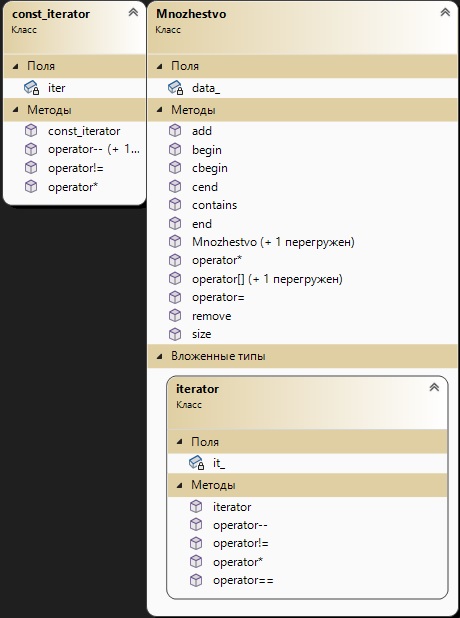
В С++ массивы поддерживают доступ по индексу. Доступ по индексу - это прямой доступ.

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

В С++ в линейной списке нет доступа по индексу, и требуется пройти по каждому элементу. Это последовательный доступ.

**Приложения**

Приложение UML-диаграмма



Приложение Б – код программы

Main.cpp:

#include <iostream>

#include <vector>

#include <algorithm>

#include <queue>

#include "Const\_iterator.h"

#include "Mnozhestvo.h"

using namespace std;

int main() {

Mnozhestvo m1, m2, m3;

m1.add(1);

m1.add(2);

m1.add(3);

m1.add(4);

m2.add(3);

m2.add(4);

m2.add(5);

m2.add(6);

m3 = m1 \* m2;

cout << "Elements of m1: ";

for (int i = 0; i < m1.size(); i++) {

cout << m1[i] << " ";

}

cout << endl;

cout << "Elements of m2: ";

for (int i = 0; i < m2.size(); i++) {

cout << m2[i] << " ";

}

cout << endl;

cout << "Elements of m3: ";

for (int i = 0; i < m3.size(); i++) {

cout << m3[i] << " ";

}

cout << endl;

cout << "Size of vector: " << m1.size() << endl;

Mnozhestvo::iterator it = m3.end();

--it;

cout << "Previous element of m3: " << \*it << endl;

return 0;

}

Mnozhestvo.h:

#pragma once

#include <iostream>

#include <vector>

#include <algorithm>

#include <queue>

#include "Const\_iterator.h"

class Mnozhestvo {

private:

vector<int> data\_;

public:

// Возвращает размер множества

int size() const {

return data\_.size();

}

// Итератор для перебора элементов множества

const\_iterator cbegin() const {

return const\_iterator(data\_.cbegin());

}

const\_iterator cend() const {

return const\_iterator(data\_.cend());

}

// конструктор по умолчанию

Mnozhestvo() {}

// конструктор копирования

Mnozhestvo(const Mnozhestvo& other) : data\_(other.data\_) {}

// оператор присваивания

Mnozhestvo& operator=(const Mnozhestvo& other) {

if (this != &other) {

data\_ = other.data\_;

}

return \*this;

}

// методы-модификаторы

void add(int value) {

if (!contains(value)) {

data\_.push\_back(value);

}

}

void remove(int value) {

data\_.erase(std::remove(data\_.begin(), data\_.end(), value), data\_.end());

}

// методы-запросы

bool contains(int value) const {

return find(data\_.begin(), data\_.end(), value) != data\_.end();

}

// операция последовательного доступа

int& operator[](int index) {

return data\_[index];

}

const int& operator[](int index) const {

return data\_[index];

}

// операция пересечения множеств

Mnozhestvo operator\*(const Mnozhestvo& other) const {

Mnozhestvo result;

for (int i = 0; i < size(); i++) {

if (other.contains((\*this)[i])) {

result.add((\*this)[i]);

}

}

return result;

}

// вложенный класс итератора

class iterator {

public:

iterator(vector<int>::iterator it) : it\_(it) {}

int& operator\*() {

return \*it\_;

}

iterator& operator--() {

--it\_;

return \*this;

}

bool operator==(const iterator& other) const {

return it\_ == other.it\_;

}

bool operator!=(const iterator& other) const {

return !(\*this == other);

}

private:

vector<int>::iterator it\_;

};

// методы, возвращающие итераторы

iterator begin() {

return iterator(data\_.begin());

}

iterator end() {

return iterator(data\_.end());

}

};

ostream& operator<<(ostream& out, const Mnozhestvo& set) {

out << "{";

for (int i = 0; i < set.size(); i++) {

out << set[i];

if (i != set.size() - 1) {

out << ", ";

}

}

out << "}";

return out;

}

Const\_iterator.h:

#pragma once

#include <iostream>

#include <vector>

#include <algorithm>

class const\_iterator {

vector<int>::const\_iterator iter;

public:

const\_iterator(vector<int>::const\_iterator iter) : iter(iter) {}

const\_iterator operator--(int) {

const\_iterator prev = \*this;

iter--;

return prev;

}

const\_iterator& operator--() {

--iter;

return \*this;

}

int operator\*() const {

return \*iter;

}

bool operator!=(const const\_iterator& other) const {

return iter != other.iter;

}

};