Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

Пермский национальный исследовательский политехнический университет

Электротехнический факультет

Кафедра информационных технологий и автоматизированных систем

**ОТЧЕТ**

**о работе по информатике**

Семестр: 2

На тему: Лабораторная работа №11: «Последовательные контейнеры библиотеки STL».

**Вариант 13**

Выполнил студент ИВТ-22-2б:

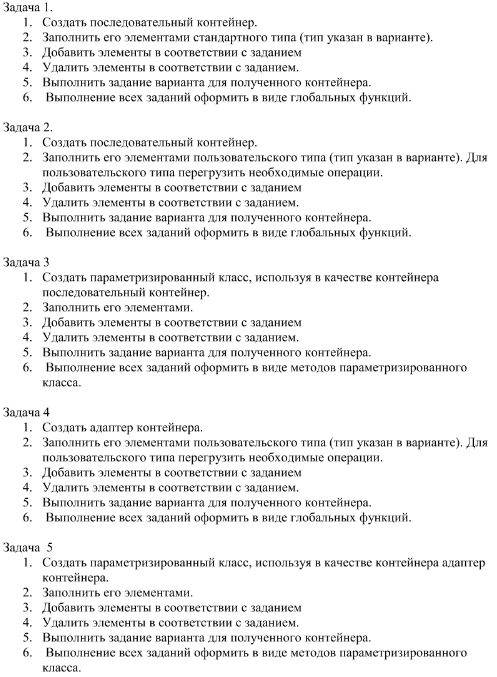
Коняев Александр Сергеевич

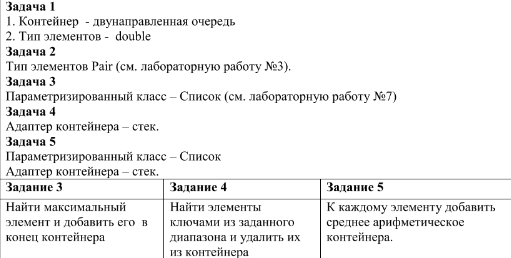
Проверил доцент кафедры ИТАС:

Полякова Ольга Андреевна

Пермь 2023

**Постановка задачи**





**Анализ задачи**

Класс Pair

* Конструктор без параметров
* Конструктор с параметрами
* Конструктор копирования
* Метод get\_first
* Метод set\_first
* Метод get\_second
* Метод set\_second
* Перегруженные операторы: =, <, >, --, <<, >>

Класс LinkedList

* Конструктор с параметрами
* Конструктор без параметров
* Конструктор копирования
* Метод clear
* Метод push\_back
* Метод push\_front
* Метод pop\_back
* Метод pop\_front
* Метод get\_At
* Метод insert
* Метод erase
* Метод print
* Перегруженные операторы: =, [], +, <<, >>

STL библиотеки: deque, stack

**UML – диаграмма**

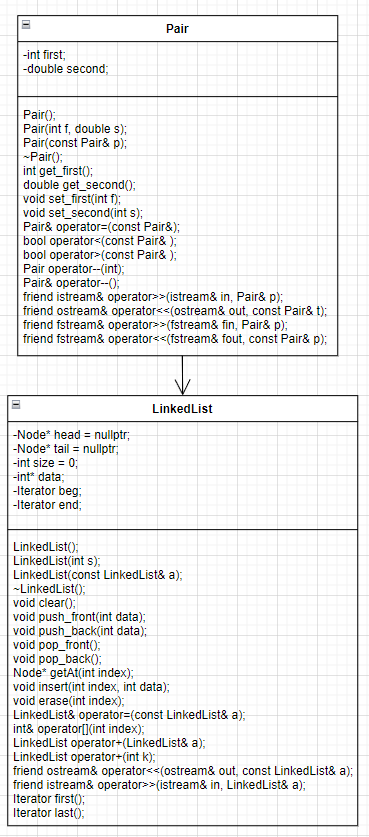


Рисунок 1 – UML-диаграмма.

**Код программы.**

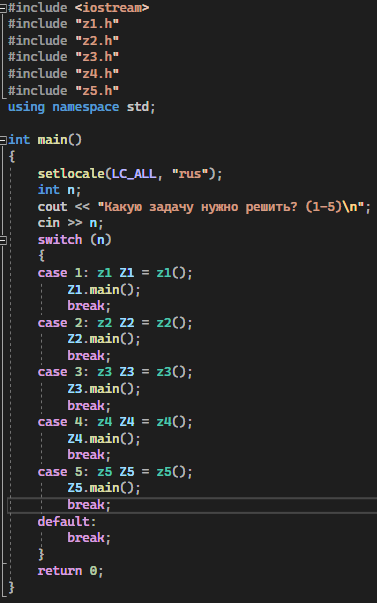


Рисунок 2 – Функция main

**Вывод программы.**

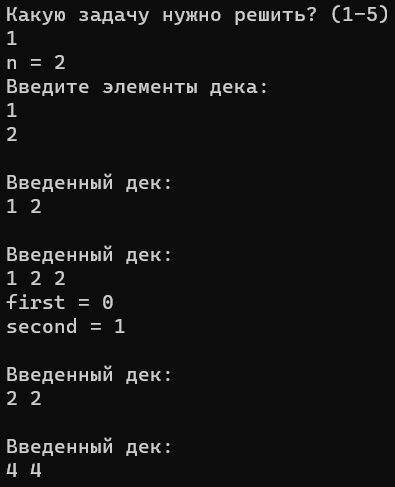
****

Рисунок 3 – Вывод программы

**Ответы на вопросы**

1. Библиотека STL (Standard Template Library) состоит из трех осных частей:

Контейнеры (containers), которые представляют собой различные типы хранилищ объектов.

Итераторы (iterators), которые используются для обхода элементов контейнеров.

Алгоритмы (algorithms), которые применяются к контейнерам и используются для выполнения различных операций над объектами в контейнерах.

1. В STL существует несколько типов контейнеров. Все они предназначены для хранения и управления коллекциями объектов. Некоторые из них перечислены ниже:

Векторы (vectors): динамический массив, который позволяет быстро добавлять или удалять элементы в конце контейнера.

Списки (lists): двунаправленный связанный список, который позволяет быстро добавлять или удалять элементы в начале, конце или середине списка.

Контейнеры ассоциативных массивов: map, multimap, set и multiset. Они позволяют быстро находить элементы по ключу и облегчают доступ к данным.

Очереди (queue): позволяют управлять элементами, использующими первый вошел, первый вышел (FIFO).

Стеки (stack): позволяют управлять элементами, использующими последний вошел, первый вышел (LIFO).

1. Для использования контейнеров STL в своей программе нужно включить заголовочный файл <vector> или <list> (в зависимости от выбранного контейнера) в начале программы. Например, для использования вектора (vector) нужно включить заголовочный файл следующим образом:

#include <vector>

1. Итератор - это объект, который предоставляет доступ к элементам контейнера и позволяет перебирать их в цикле. Итераторы могут быть использованы для чтения, записи и обхода элементов контейнера. Они предоставляют стандартизованный интерфейс для работы с контейнерами, независимый от их реализации.
2. Операции, которые можно выполнять над итераторами, зависят от конкретного типа итератора и контейнера, к которому он относится. Но в целом, итераторы обеспечивают доступ к элементам контейнера, перебор их в цикле, поддерживают арифметику указателей для перемещения по контейнеру и позволяют выполнить следующие операции:

Доступ к текущему элементу с помощью оператора разыменования (\*)

Получение адреса текущего элемента с помощью оператора взятия адреса (&)

Перемещение указателя на следующий или предыдущий элемент контейнера с помощью операторов инкремента (++) и декремента (--)

Сравнение итераторов на равенство или неравенство

Выполнение арифметических операций, таких как сложение и вычитание, на итераторах для перемещения указателя на несколько элементов вперед или назад.

1. Цикл для перебора контейнера с использованием итератора можно организовать следующим образом:

// Создание итератора и задание начальной позиции

std::vector<int>::iterator it = vec.begin();

// Перебор контейнера с использованием итератора в цикле

for (; it != vec.end(); ++it) {

// Доступ к текущему элементу с помощью оператора разыменования

int element = \*it;

// Операции с текущим элементом

std::cout << element << " ";

}

std::cout << std::endl;

В этом коде создается итератор для вектора и задается начальная позиция в начало вектора с помощью функции begin(). Затем происходит перебор контейнера с помощью итератора в цикле до тех пор, пока итератор не достигнет конца контейнера (end()). В теле цикла из итератора извлекается текущий элемент контейнера с помощью оператора разыменования (\*it), после чего можно выполнять необходимые операции с текущим элементом.

1. Существует пять типов итераторов в стандартной библиотеке C++:

Input Iterator (Входной итератор) - используется для перемещения вперед по контейнеру, позволяет только чтение данных.

Output Iterator (Выходной итератор) - используется для перемещения вперед по контейнеру, позволяет только запись данных.

Forward Iterator (Прямой итератор) - используется для перемещения вперед и обратно по контейнеру, позволяет как чтение, так и запись данных.

Bidirectional Iterator (Двунаправленный итератор) - используется для перемещения вперед и назад по контейнеру, позволяет как чтение, так и запись данных.

Random Access Iterator (Итератор произвольного доступа) - используется для перемещения по контейнеру с помощью операторов [] и +-, позволяет как чтение, так и запись данных.

1. В стандартной библиотеке C для всех контейнеров доступны следующие операции и методы:

size() - возвращает количество элементов в контейнере

empty() - проверяет, является ли контейнер пустым

begin() - возвращает итератор на первый элемент контейнера

end() - возвращает итератор на элемент, следующий за последним элементом контейнера

rbegin() - возвращает итератор на последний элемент контейнера

rend() - возвращает итератор на элемент, предшествующий первому элементу контейнера

clear() - удаляет все элементы контейнера

erase() - удаляет элемент или диапазон элементов из контейнера

insert() - вставляет элемент или диапазон элементов в контейнер

swap() - обменивает содержимое двух контейнеров

operator==() - проверяет два контейнера на равенство

operator!=() - проверяет два контейнера на неравенство

operator<() - сравнивает два контейнера на меньше

operator>() - сравнивает два контейнера на больше

operator<=() - сравнивает два контейнера на меньше или равно

operator>=() - сравнивает два контейнера на больше или равно

1. Контейнер vector поддерживает эффективную операцию доступа к элементу по индексу (operator[]) и добавление новых элементов в конец контейнера (push\_back()). Эти операции выполняются за время O(1) в среднем случае. Однако, операции вставки и удаления элементов из середины контейнера выполняются за время O(n), так как требуется сдвиг всех элементов после удаляемого/вставляемого. Поэтому, если операции вставки и удаления являются основными, то контейнеры списков (list) или деков (deque) могут быть более эффективными выборами.
2. Контейнер list поддерживает эффективные операции вставки и удаления элементов в середине списка, так как для этого не требуется сдвигать все элементы после удаляемого/вставляемого, как это происходит в контейнере vector. Операции вставки и удаления элемента в середине списка выполняются за время O(1). Однако, доступ к элементам по индексу в контейнере list происходит за время O(n), так как список не поддерживает прямой доступ к элементу по индексу, что делает контейнер vector более эффективным в случае частого обращения по индексу.
3. Контейнер deque (двусторонняя очередь) поддерживает эффективные операции вставки и удаления элементов на обоих концах очереди, так как для этого не требуется сдвигать все элементы, как в случае с контейнером vector. Операции вставки и удаления элемента в начале и конце deque выполняются за время O(1) в среднем случае. Однако, доступ к элементам по индексу в контейнере deque также происходит за время O(1) в среднем случае. Это происходит благодаря тому, что deque использует массив блоков в качестве внутренней реализации, что позволяет обращаться к элементам в любой момент времени без необходимости перемещать остальные элементы. Таким образом, deque сочетает эффективные операции вставки/удаления в начале и конце и быстрый прямой доступ к элементам по индексу, что делает его универсальным контейнером для многих задач.
4. Методы, поддерживаемые последовательным контейнером vector:

push\_back

pop\_back

clear

insert

erase

resize

reserve

size

capacity

empty

operator[]

at

front

back

data

Некоторые из этих методов используются для добавления, удаления или изменения элементов в контейнере, в то время как другие методы возвращают информацию об элементах в контейнере.

1. Методы, поддерживаемые последовательным контейнером list:

push\_back

push\_front

pop\_back

pop\_front

emplace\_front

emplace\_back

emplace

clear

insert

erase

resize

reverse

size

max\_size

empty

front

back

assign

remove

remove\_if

splice

merge

unique

sort

swap

get\_allocator

1. Методы, которые поддерживает последовательный контейнер deque:

push\_back push\_front pop\_back pop\_front emplace\_front emplace\_back emplace clear insert erase resize swap at front back begin end rbegin rend cbegin cend crbegin crend size max\_size empty shrink\_to\_fit get\_allocator

1. #include <vector>

std::vector<int> myVec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

myVec.erase(myVec.begin() + 2, myVec.begin() + 6);

// Теперь вектор содержит элементы {1, 2, 7, 8, 9, 10}

1. #include <vector>

std::vector<int> myVec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

myVec.pop\_back(); // удаляем последний элемент

// Теперь вектор содержит элементы {1, 2, 3, 4, 5, 6, 7, 8, 9}

1. #include <list>

std::list<int> myList = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

auto start = std::next(myList.begin(), 1); // итератор на второй элемент (индексация с нуля)

auto end = std::next(myList.begin(), 5); // итератор на шестой элемент

myList.erase(start, end); // удаляем элементы со 2 по 5

// Теперь в списке содержатся элементы {1, 6, 7, 8, 9, 10}

1. std::list<int> myList = {1, 2, 3, 4, 5};

myList.pop\_back();

1. #include <iostream>

#include <deque>

int main() {

std::deque<int> myDeque = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::deque<int>::iterator it = myDeque.begin() + 1; // итератор указывает на второй элемент

myDeque.erase(it, it + 4); // удаляем элементы со 2 по 5 (включительно)

// печатаем оставшиеся элементы

for (auto elem : myDeque) {

std::cout << elem << " ";

}

std::cout << std::endl;

return 0;

}

1. #include <iostream>

#include <deque>

int main() {

std::deque<int> myDeque = {1, 2, 3, 4, 5};

myDeque.pop\_back(); // удаляем последний элемент

// печатаем оставшиеся элементы

for (auto elem : myDeque) {

std::cout << elem << " ";

}

std::cout << std::endl;

return 0;

}

1. template <typename T>

void printContainer(const T& container) {

for (auto it = container.begin(); it != container.end(); ++it) {

std::cout << \*it << " ";

}

std::cout << std::endl;

}

1. Адаптеры контейнеров - это классы, которые предоставляют доступ к элементам контейнеров с помощью различных интерфейсов. Они представляют собой интерфейсы, которые позволяют работать с контейнерами в определенных условиях, таких как стек (LIFO), очередь (FIFO), а также оставлять только уникальные элементы или отсортировать элементы в порядке возрастания или убывания.

Адаптеры контейнеров обычно создаются путем использования декоратора - это означает, что класс-адаптер использует экземпляр из оригинального контейнера в качестве переменной-члена для доступа к элементам. Это делает адаптеры контейнеров эффективными способами получения доступа к элементам других контейнеров с использованием различных интерфейсов, не требуя переписывания или перекомпиляции существующего кода.

1. Объявление stack<int> s создает экземпляр стека, который использует встроенный контейнер std::deque, тогда как объявление stack<int, list<int> > s создает экземпляр стека, который использует контейнер std::list.

То есть, первый вариант будет использовать двустороннюю очередь, а второй - связный список. Это означает, что возможности и особенности доступа к элементам в каждом случае будут отличаться и зависеть от особенностей соответствующего контейнера.

1. Контейнер stack поддерживает следующие методы:

empty() - проверяет, является ли стек пустым

size() - возвращает количество элементов в стеке

top() - возвращает ссылку на верхний элемент стека

push(const T& value) - добавляет элемент на вершину стека

emplace(Args&&... args) - создает и добавляет элемент на вершину стека с использованием переданных аргументов

pop() - удаляет элемент с вершины стека

Кроме того, stack наследует от контейнера deque или list все методы, которые поддерживают эти контейнеры.

1. Контейнер queue поддерживает следующие методы:

empty() - проверяет, является ли очередь пустой

size() - возвращает количество элементов в очереди

front() - возвращает ссылку на первый элемент в очереди

back() - возвращает ссылку на последний элемент в очереди

push(const T& value) - добавляет элемент в конец очереди

emplace(Args&&... args) - создает и добавляет элемент в конец очереди с использованием переданных аргументов

pop() - удаляет первый элемент из очереди

1. Контейнер queue представляет стандартную очередь, в которой элемент, добавленный первым, будет удален первым (FIFO). Priority\_queue представляет очередь с приоритетом, в которой каждый элемент имеет свой приоритет и элемент с более высоким приоритетом стоит в начале очереди. При удалении элементов из priority\_queue элемент с наибольшим приоритетом будет удален первым. Другое отличие заключается в том, что в priority\_queue метод push добавляет элемент в очередь в порядке приоритета, а не просто в конец.
2. #include <stack>

#include <iostream>

int main() {

std::stack<int> my\_stack;

my\_stack.push(1);

my\_stack.push(2);

my\_stack.push(3);

my\_stack.push(4);

my\_stack.push(5);

int index\_to\_remove = 3;

int current\_index = 1;

std::stack<int> temp\_stack;

while (!my\_stack.empty()) {

int current\_element = my\_stack.top();

my\_stack.pop();

if (current\_index != index\_to\_remove) {

temp\_stack.push(current\_element);

}

current\_index++;

}

while (!temp\_stack.empty()) {

int current\_element = temp\_stack.top();

temp\_stack.pop();

my\_stack.push(current\_element);

}

// Печатаем исходный стек после удаления элемента

while (!my\_stack.empty()) {

int current\_element = my\_stack.top();

my\_stack.pop();

std::cout << current\_element << " ";

}

return 0;

}

1. #include <queue>

#include <iostream>

int main() {

std::queue<int> my\_queue;

my\_queue.push(1);

my\_queue.push(2);

my\_queue.push(3);

my\_queue.push(4);

my\_queue.push(5);

int index\_to\_remove = 3;

int current\_index = 1;

std::queue<int> temp\_queue;

while (!my\_queue.empty()) {

int current\_element = my\_queue.front();

my\_queue.pop();

if (current\_index != index\_to\_remove) {

temp\_queue.push(current\_element);

}

current\_index++;

}

while (!temp\_queue.empty()) {

int current\_element = temp\_queue.front();

temp\_queue.pop();

my\_queue.push(current\_element);

}

// Печатаем исходную очередь после удаления элемента

while (!my\_queue.empty()) {

int current\_element = my\_queue.front();

my\_queue.pop();

std::cout << current\_element << " ";

}

return 0;

}

1. #include <iostream>

#include <stack>

template<typename T>

void printStack(std::stack<T> s) {

while (!s.empty()) {

std::cout << s.top() << " ";

s.pop();

}

std::cout << std::endl;

}

int main() {

std::stack<int> myStack;

myStack.push(1);

myStack.push(2);

myStack.push(3);

myStack.push(4);

myStack.push(5);

printStack(myStack);

return 0;

}

1. #include <iostream>

#include <queue>

template<typename T>

void printQueue(std::queue<T> q) {

while (!q.empty()) {

std::cout << q.front() << " ";

q.pop();

}

std::cout << std::endl;

}

int main() {

std::queue<int> myQueue;

myQueue.push(1);

myQueue.push(2);

myQueue.push(3);

myQueue.push(4);

myQueue.push(5);

printQueue(myQueue);

return 0;

}