**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ**

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Кафедра «Информационная безопасность»

Лабораторная работа №4

по дисциплине «Разработка безопасного программного обеспечения»

Классы с динамическими структурами данных.

**Вариант 10**

Выполнили студенты группы: БАС 2101

Минеев Никита Сергеевич

Руководитель: Барков В.В.

Москва 2023

### Цель работы

Изучить динамические структуры данных, овладеть навыками создания конструкторов копирования, перемещения, деструкторов, перегрузки операций копирования и перемещения.

### Краткие теоретические сведения

При разработке классов, содержащих ссылки на какие-либо ресурсы, будь то динамически выделенная память или другие ресурсы операционной системы, возникают проблемы при копировании объектов. Обычно ресурсы захватываются в конструкторе класса и освобождаются в деструкторе. Автоматически генерируемый конструктор копирования выполняет почленное копирование. В этом случае после копирования два объекта будут иметь ссылку на один и тот же ресурс и в момент удаления каждый из них попытается его освободить. Уже после того как первый объект освободит ресурс, второй объект будет работать некорректно. Но ещё хуже, когда второй объект повторно попытается освободить ресурс. Для того чтобы избежать таких ситуаций, необходимо предоставить собственную реализацию конструктора копирования.

#### Конструктор копирования и операция присваивания

Пусть имеется определение класса Stack, представляющего стек на основе массива:

|  |  |
| --- | --- |
| class Stack  {  public:  Stack(int count);  Stack(const Stack &other);  Stack(Stack &&other);  Stack &operator=(const Stack &other);  ~Stack();  int GetSize() const;  void Push(int element);  int Pop();  private:  int \*arr;  int count;  };  Stack::Stack(int count)  : count(count)  {  arr = new int[count];  }  Stack::~Stack()  {  delete[] arr;  } | Stack::Stack(const Stack &other)  {  arr = new int[other.count];  count = other.count;  for (int i = 0; i < count; ++i)  {  arr[i] = other.arr[i];  }  }  Stack &Stack::operator=(const Stack &other)  {  if (this == &other)  {  return \*this;  }  delete[] arr;  arr = new int[other.count];  count = other.count;  for (int i = 0; i < count; ++i)  {  arr[i] = other.arr[i];  }  return \*this;  } |

Конструктор копирования для этого класса выполняет следующее: cначала выделяет объем памяти, достаточный для копирования элементов массива, а потом собственно копирует элементы.

При таком подходе описанная выше проблема исчезает для случаев инициализации объекта, однако это не решает проблему при присваивании одного объекта другому. Причиной этого является то, что операция присваивания, автоматически генерируемая для класса, также выполняет почленное копирование. Для того чтобы исправить ситуацию необходимо явно определить операцию присваивания.

Логика операции присваивания немного сложнее, поскольку прежде чем копировать данные, необходимо освободить ресурсы, например, выделенную память (по сути повторить логику деструктора), а потом повторить логику конструктора копирования. Для того чтобы безвозвратно не потерять данные при присваивании объекта самому себе, необходимо отдельно обработать этот случай.

Можно избежать дублирование кода. Деструктор можно вызвать явно, а логику конструктора копирования выделить в отдельную приватную функцию.

#### Конструктор перемещения и операция присваивания с перемещением

В стандарте C++11 появились конструктор перемещения и операция присваивания с перемещением. Они призваны улучшить производительность копирования, когда объект-источник является временным объектом. Вместо того, чтобы выделять новую память и копировать данные можно стать владельцем уже выделенной, однако при этом временный объект должен перестать владеть ресурсом. В примере с динамической памятью мы должны обнулить указатель.

|  |  |
| --- | --- |
| Конструктор перемещения | Операция присваивания с перемещением |
| Stack::Stack(Stack &&other)  {  arr = other.arr;  count = other.count;  other.arr = nullptr;  } | Stack &Stack::operator=(Stack &&other)  {  if (this == &other)  {  return \*this;  }  delete[] arr;  arr = other.arr;  count = other.count;  other.arr = nullptr;  } |

### Задание

Для типа динамической структуры данных, указанного в индивидуальном задании, разработать соответствующий класс, предусмотрев в нем:

конструктор инициализации,

конструктор копирования

конструктор перемещения,

деструктор,

функции-члены, указанные в таблице.

|  |  |
| --- | --- |
|  | Динамический массив DynamicArray |
| Конструкторы и деструктор | DynamicArray(); explicit DynamicArray(unsigned size);  DynamicArray(const DynamicArray& other);  DynamicArray(DynamicArray&& other);  DynamicArray(std::initializer\_list<int> list);  ~DynamicArray(); |
| Функции-члены | unsigned GetSize() const;  DynamicArray::Iterator begin();  const DynamicArray::Iterator begin() const;  DynamicArray::Iterator end();  const DynamicArray::Iterator end() const; |
| Перегруженные операции | const int& operator[](int index) const;  int& operator[](int index);  DynamicArray& operator=(const DynamicArray& other);  DynamicArray& operator=(DynamicArray&& other);  DynamicArray& operator=( std::initializer\_list<int> ilist); |
|  |  |
|  |  |

|  |  |  |
| --- | --- | --- |
|  | Динамический массив | Связный список |
| Конструкторы и деструктор | Конструктор по умолчанию, Явный конструктор преобразования  Конструктор копирования  Конструктор перемещения  Конструктор, принимающий на вход std::initializer\_list,  деструктор | Конструктор по умолчанию, Конструктор копирования  Конструктор перемещения  Конструктор, принимающий на вход std::initializer\_list,  деструктор |
| Функции-члены | unsigned GetSize() const; | unsigned GetSize() const;  void InsertAt(int value, int index);  int RemoveAt(int index); |
| Перегруженные операции | const int& operator[](int index) const;  int& operator[](int index);  List& operator=(const List& other);  List& operator=(List&& other);  List& operator=( std::initializer\_list<int> ilist); | List& operator=(const List& other);  List& operator=(List&& other);  List& operator=( std::initializer\_list<int> ilist); |
|  |  |  |
|  |  |  |

Перегрузить операции присваивания, присваивание с перемещением и потокового вывода для вывода содержимого динамической структуры на экран. Обязательные функции-члены:

|  |  |  |
| --- | --- | --- |
| Стек | Очередь | Дек |
| int GetSize() const;  bool Push(const T &element);  bool Pop(T &element);  bool Peek(T &element); | int GetSize();  bool Push(const T &element);  bool Pop(T &element);  bool Peek(T &element); | int GetSize() const;  bool PushFront(const T &element);  bool PopFront(T &element);  bool PushBack(const T &element);  bool PopBack(T &element);  bool PeekFront(T &element) const;  bool PeekBack(T &element) const; |

Функции Push, PushFront и PushBack должны возвращать true в случае успешного добавления и false в случае нехватки места.

Функции Pop, PopBack, PopFront, Peek, PeekBack, PeekFront, должны возвращать true и копию элемента через параметр element, передаваемый по ссылке, в случае если в контейнере есть элементы и false, в случае если в контейнере нет элементов.

### Индивидуальное задание

|  |  |  |
| --- | --- | --- |
| 3 | Стек на основе двунаправленного списка | StackBasedOnBidirectionalLinkedList |

**Реализация кода:**

#include <iostream>

#include <initializer\_list>

//Очередь на основе двунаправленного циклического списка

template < typename T >

struct Element {

T number;

Element\* next;

Element\* prev;

};

template < typename T > class QueueBasedOnBidirectionalCyclicLinkedList {

private: Element < T >\* head;

Element < T >\* tail;

public:

//конструктор по умолчанию

QueueBasedOnBidirectionalCyclicLinkedList();

//конструктор копирования

QueueBasedOnBidirectionalCyclicLinkedList(const QueueBasedOnBidirectionalCyclicLinkedList& \_object);

//конструктор перемещения

QueueBasedOnBidirectionalCyclicLinkedList(QueueBasedOnBidirectionalCyclicLinkedList&& \_object);

//конструктор с использованием списка инициализации

QueueBasedOnBidirectionalCyclicLinkedList(std::initializer\_list < int > \_ilist);

//деконструктор

~QueueBasedOnBidirectionalCyclicLinkedList();

//перегрузка оператора присваивания для копирования

QueueBasedOnBidirectionalCyclicLinkedList& operator = (const QueueBasedOnBidirectionalCyclicLinkedList& \_object);

//перегрузка оператора присваивания для перемещения

QueueBasedOnBidirectionalCyclicLinkedList& operator = (QueueBasedOnBidirectionalCyclicLinkedList&& \_object);

//перегрузка оператора присваивания для списка инициализации

QueueBasedOnBidirectionalCyclicLinkedList& operator = (std::initializer\_list < int > \_ilist);

//потоковый вывод

template < typename T >

friend std::ostream& operator << (std::ostream& stream,

const QueueBasedOnBidirectionalCyclicLinkedList < T >& \_object);

//функции члены

//получение размера

int GetSize();

void InsertAt(int \_value, int \_index);

int RemoveAt(int \_index);

//Методы Push, Pop, Peek

bool Push(const T& element);

bool Pop(T& element);

bool Peek(T& element);

};

//конструктор по умолчанию

template < typename T >

QueueBasedOnBidirectionalCyclicLinkedList < T > ::QueueBasedOnBidirectionalCyclicLinkedList() {

head = tail = NULL;

}

//консруктор копирования

template < typename T >

QueueBasedOnBidirectionalCyclicLinkedList < T > ::QueueBasedOnBidirectionalCyclicLinkedList(const QueueBasedOnBidirectionalCyclicLinkedList& \_object) {

if (\_object.head == NULL) {

head = tail = NULL;

return;

}

Element < T >\* temp = \_object.head;

while (temp != \_object.tail) {

this->Push(temp->number);

temp = temp->prev;

}

this->Push(temp->number);

}

//конструктор перемещения

template < typename T >

QueueBasedOnBidirectionalCyclicLinkedList < T > ::QueueBasedOnBidirectionalCyclicLinkedList(QueueBasedOnBidirectionalCyclicLinkedList&& \_object) {

if (\_object.head == NULL) {

head = tail = NULL;

return;

}

head = \_object.head;

tail = \_object.tail;

\_object.head = \_object.tail = NULL;

}

//конструктор для листа инициализации

template < typename T >

QueueBasedOnBidirectionalCyclicLinkedList < T > ::QueueBasedOnBidirectionalCyclicLinkedList(std::initializer\_list < int > \_ilist) {

for (const auto& element : \_ilist)

this->Push(element);

}

//деструктор

template < typename T >

QueueBasedOnBidirectionalCyclicLinkedList < T > ::~QueueBasedOnBidirectionalCyclicLinkedList() {

if (head == NULL) return;

int temp;

while (this->GetSize() > 0) {

this->Pop(temp);

}

head = tail = NULL;

}

//операция присвоение

template < typename T >

QueueBasedOnBidirectionalCyclicLinkedList < T >& QueueBasedOnBidirectionalCyclicLinkedList < T > ::operator = (const QueueBasedOnBidirectionalCyclicLinkedList& \_object) {

if (this == &\_object || \_object.head == NULL)

return \*this;

int pop;

for (int size = this->GetSize(); size > 0; size--)

this->Pop(pop);

Element < T >\* temp = \_object.head;

while (temp != \_object.tail) {

this->Push(temp->number);

temp = temp->prev;

}

this->Push(temp->number);

return \*this;

}

//операция присвоение с перемещением

template < typename T >

QueueBasedOnBidirectionalCyclicLinkedList < T >& QueueBasedOnBidirectionalCyclicLinkedList < T > ::operator = (QueueBasedOnBidirectionalCyclicLinkedList&& \_object) {

if (this == &\_object || \_object.head == NULL)

return \*this;

head = \_object.head;

tail = \_object.tail;

\_object.head = \_object.tail = NULL;

return \*this;

}

template < typename T >

QueueBasedOnBidirectionalCyclicLinkedList < T >& QueueBasedOnBidirectionalCyclicLinkedList < T > ::operator = (std::initializer\_list < int > \_ilist) {

int temp;

for (int size = this->GetSize(); size > 0; size--)

this->Pop(temp);

for (const auto& element : \_ilist)

this->Push(element);

return \*this;

}

//получение размера

template < typename T >

int QueueBasedOnBidirectionalCyclicLinkedList < T > ::GetSize() {

int i = 0;

Element < T >\* temp = tail;

if (head == NULL) return 0;

while (temp != head) {

i++;

temp = temp->next;

}

return i + 1;

}

template < typename T >

void QueueBasedOnBidirectionalCyclicLinkedList < T > ::InsertAt(int \_value, int \_index) {

int size = this->GetSize() + 1;

QueueBasedOnBidirectionalCyclicLinkedList < int >\* temp\_obj{

std::move(this)

};

int tmp;

if (\_index <= size)

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

if (\_index == i) {

this->Push(\_value);

continue;

}

temp\_obj->Pop(tmp);

this->Push(tmp);

}

}

template < typename T >

int QueueBasedOnBidirectionalCyclicLinkedList < T > ::RemoveAt(int \_index) {

int size = this->GetSize();

QueueBasedOnBidirectionalCyclicLinkedList < int >\* temp\_obj{

std::move(this)

};

int removedElement;

int tmp;

if (\_index <= size)

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

if (\_index == i) {

temp\_obj->Pop(removedElement);

continue;

}

temp\_obj->Pop(tmp);

this->Push(tmp);

}

return removedElement;

}

// Метод для добавления элемента в конец очереди

template < typename T >

bool QueueBasedOnBidirectionalCyclicLinkedList < T > ::Push(const T& element) {

Element < T >\* temp = new Element < int >;

temp->number = element;

if (head != NULL) {

temp->next = tail;

temp->prev = head;

tail->prev = temp;

head->next = temp;

tail = temp;

}

else temp->next = temp->prev = head = tail = temp;

return true;

}

// Метод для удаления элемента из начала очереди

template < typename T >

bool QueueBasedOnBidirectionalCyclicLinkedList < T > ::Pop(T& element) {

if (head == tail) {

if (head == NULL) return false;

element = head->number;

delete head;

head = tail = NULL;

return true;

}

element = head->number;

head = head->prev;

delete head->next;

head->next = tail;

tail->prev = head;

return true;

}

// Метод для просмотра элемента в начале очереди

template < typename T >

bool QueueBasedOnBidirectionalCyclicLinkedList < T > ::Peek(T& element) {

if (head == NULL) return false;

element = head->number;

return true;

}

//перегруженный оператр вывода

template < typename T >

std::ostream& operator << (std::ostream& stream,

const QueueBasedOnBidirectionalCyclicLinkedList < T >& \_object) {

if (\_object.head == NULL) return stream << "Queue is empty!";

Element < T >\* temp = \_object.head;

while (temp != \_object.tail) {

stream << temp->number << " ";

temp = temp->prev;

}

return stream << temp->number << std::endl;

}

int main() {

int peekElement = 0;

int popElement = 0;

QueueBasedOnBidirectionalCyclicLinkedList < int > default\_queue; // Конструктор по умолчанию

QueueBasedOnBidirectionalCyclicLinkedList < int > initialization\_queue = {

1,

2,

3,

4,

5

}; // Конструктор с использованием списка инициализации

QueueBasedOnBidirectionalCyclicLinkedList < int > copy\_queue{

initialization\_queue

}; // Конструктор кпирования

QueueBasedOnBidirectionalCyclicLinkedList < int > temp\_queue{

copy\_queue

}; // Конструктор кпирования

QueueBasedOnBidirectionalCyclicLinkedList < int > move\_queue{

std::move(temp\_queue)

}; //конструктор переноса

//провера вывода GetSize()

std::cout << "Size of default\_queue default constructor: " << default\_queue.GetSize() << std::endl;

std::cout << "Size of initialization\_queue initialization\_list: " << initialization\_queue.GetSize() << std::endl;

std::cout << "Size of copy\_queue copy constructor: " << copy\_queue.GetSize() << std::endl;

std::cout << "Size of move\_queue move constructor: " << move\_queue.GetSize() << std::endl << std::endl;

//проверка вывода через переруженный оператор потокового вывода

std::cout << "Elements of initialization list: " << initialization\_queue << std::endl;

//тестирование заполнения масива

default\_queue = {

1,

2,

3,

4

};

copy\_queue = {

2,

3,

4

};

//вывод элементов заполненых массивов

std::cout << "filled new elements of default\_queue: ";

std::cout << default\_queue;

std::cout << "filled new elements of copy\_queue: ";

std::cout << copy\_queue << std::endl;

//проверка перегруженных операторов Pop, Push, Peek

std::cout << "Push >> Peek >> Pop:" << std::endl;

std::cout << "All elements of default\_queue before Push: ";

std::cout << default\_queue;

default\_queue.Push(10);

std::cout << "All elements of default\_queue after Push: ";

std::cout << default\_queue;

std::cout << "Peeked element of default\_queue: ";

default\_queue.Peek(peekElement);

std::cout << peekElement << std::endl;

default\_queue.Pop(popElement);

std::cout << "Popped back element of default\_queue: ";

std::cout << popElement << std::endl;

std::cout << "All elements of default\_queue after Pop: ";

std::cout << default\_queue << std::endl;

//проверка методов InsertAt + RemoveAt

std::cout << "InsertAt >> RemoveAt:" << std::endl;

std::cout << "All elements of default\_queue before InsertAt(1): ";

std::cout << default\_queue;

std::cout << "All elements of default\_queue after InsertAt(1000, 1): ";

default\_queue.InsertAt(1000, 1);

std::cout << default\_queue;

std::cout << "All elements of default\_queue after RemoveAt(2): ";

default\_queue.RemoveAt(2);

std::cout << default\_queue;

return 0;

}

**Вывод**

В ходе выполнения лабораторной работы были изучены динамические структуры данных, овладены навыками создания конструкторов копирования, перемещения, деструкторов, перегрузки операций копирования и перемещения.

https://github.com/LoneGuardian1551/RPBO4LABA-MA-/tree/main

**Контрольный вопрос**

Для корректной работы классов, содержащих указатели, необходимо переопределить следующие члены:

1. Конструктор копирования. Этот конструктор используется для создания нового объекта, копируя существующий. Он должен корректно копировать все данные, включая указатели. При копировании указателей обычно производится глубокое копирование, то есть создается копия объекта, на который указывает указатель, а не сам указатель
2. Оператор присваивания. Этот оператор используется для копирования одного объекта в другой. Как и конструктор копирования, он должен корректно копировать все данные, включая указателя
3. Конструктор перемещения. Этот конструктор используется для создания нового объекта, перемещая существующий. Он должен корректно переместить все данные, включая указатели, и затем оставить исходный объект в безопасном состоянии
4. Оператор присваивания перемещением. Этот оператор используется для перемещения одного объекта в другой. Как и конструктор перемещения, он должен корректно переместить все данные, включая указатели, и затем оставить исходный объект в безопасном состоянии
5. Деструктор. Деструктор используется для освобождения ресурсов, которые были выделены объекту во время его жизни. Если объект содержит указатели на динамически выделенную память, деструктор должен корректно освободить эту память, чтобы избежать утечек памяти