Министерство образования Республики Беларусь
Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра электронных вычислительных машин

ОТЧЕТ
О ЛАБОРАТОРНОЙ РАБОТЕ № 8

Классы контейнеры-итераторы.
STL-контейнеры

по дисциплине «Программирование на языках высокого уровня»

Выполнил ст. гр. 450503 А.П. Красько

Проверил асс. каф. ЭВМ И.Г. Скиба

Минск 2025

1 ПОСТАНОВКА ЗАДАЧИ

Реализовать классы контейнер и итератор для работы с двунаправленным кольцом. Реализовать класс алгоритма, в котором определить методы поиска и сортировки двунаправленного кольца. Выполнить программу.

2 ДИАГРАММА КЛАССОВ

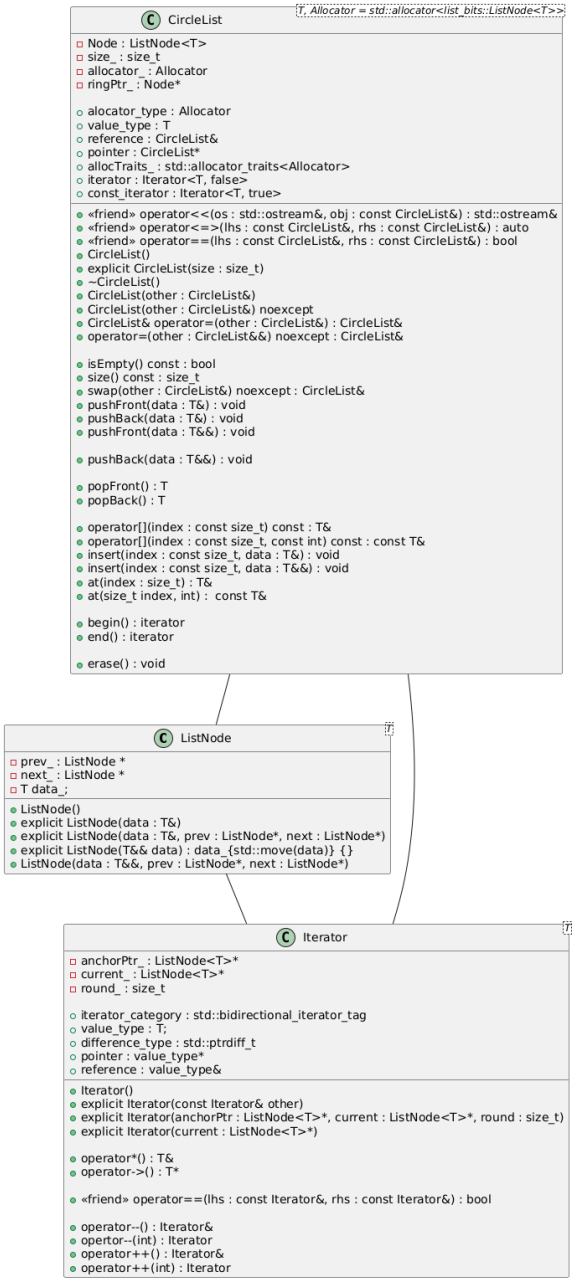

Рисунок 2.1 – Диаграмма классов

3 ЛИСТИНГ КОДА

Файл list_algorithm.hh:

```
#pragma once
#include <optional>

#include "list.hh"
namespace cList {
template <typename U, std::forward_iterator Iter>
Iter find(Iter begin, Iter end, U& data) {
    for (; begin != end; ++begin) {
        if (*begin == data) break;
    }
    return begin;
}

template <std::forward_iterator ForwardIterator>
void bubbleSort(ForwardIterator first, ForwardIterator last) {
    if (first == last) {
        return;
    }
    bool swapped;
    do {
        swapped = false;
        ForwardIterator current = first;
        ForwardIterator next_it = std::next(current);

        while (next_it != last) {
            if (*next_it < *current) {
                std::iter_swap(current, next_it);
                swapped = true;
            }
            ++current;
            ++next_it;
        }
    } while (swapped);
}
}  // namespace cList
```

Файл list_bits.hh:

```
#include <iostream>
#include <iterator>
namespace list_bits {
template <typename T>
struct ListNode {
   public:
    ListNode *prev_ = nullptr;
    ListNode *next_ = nullptr;
    T data_;
    ListNode() :  data_{T()} {}
```

```
    explicit ListNode(T& data) :  data_{data} {}
    explicit ListNode(T& data, ListNode* prev, ListNode* next) : prev_{prev},
next_{next}, data_{data} {}
    explicit ListNode(T&& data) : data_{std::move(data)} {}
    ListNode(T&& data, ListNode* prev, ListNode* next) : prev_{prev},
next_{next}, data_{std::move(data)} {}
};
template <typename T>
class Iterator  {
   private:
    ListNode<T>*anchorPtr_;
    ListNode<T> *current_;
    size_t round_ = 0;

   public:
    using iterator_category = std::bidirectional_iterator_tag;
    using value_type = T;
    using difference_type = std::ptrdiff_t;
    using pointer = value_type*;
    using reference = value_type&;

    friend Iterator<const T>;
    Iterator() = default;
    explicit Iterator(const Iterator& other) = default;
    explicit Iterator(ListNode<T>* anchorPtr, ListNode<T>* current, size_t
round)
        : anchorPtr_{anchorPtr}, current_{current}, round_{round} {}
    explicit Iterator(ListNode<T>* current) : anchorPtr_{current},
current_{current} {}

    template <typename U> requires (!std::is_const_v<U>)
    Iterator(const Iterator<U>& rhs) : anchorPtr_{rhs.anchorPtr_},
current_{rhs.current_} {}

    reference operator*() const { return this->current_->data_; }
    pointer operator->() { return &(this->current_->data_); }

    friend bool operator==(const Iterator& lhs, const Iterator& rhs)  {
return lhs.current_ == rhs.current_ && lhs.round_ == rhs.round_; }

    Iterator& operator--() {
        this->current_ = this->current_->prev_;
        if (this->current_ == this->anchorPtr_) {
            this->round_--;
        }
        return *this;
    }
    Iterator operator--(int) {
        Iterator cop{*this};
        --(*this);
        return cop;
    }
    Iterator& operator++() {
        this->current_ = this->current_->next_;
        if (this->current_ == this->anchorPtr_) {
```

```
            this->round_++;
        }
        return *this;
    }
    Iterator operator++(int) {
        Iterator cop{*this};
        ++(*this);
        return cop;
    }

};
static_assert(std::bidirectional_iterator<Iterator<int>>);
}  // namespace list_bits
```

Файл screens.hh:

```
#pragma once
#include <l8/include/list.hh>
#include <memory>

namespace screens {
void printMainScreen();
bool inputList(cList::CircleList<double> &list);
bool addElement(cList::CircleList<double> &list);
bool clearList(cList::CircleList<double> &list);
bool sortList(cList::CircleList<double> &list);
bool findElement(cList::CircleList<double> &list);
bool printList(const cList::CircleList<double> &list);
}  // namespace screens
```

Файл list.hh:

```
#pragma once
#include <execution>
#include <iostream>
#include <memory>

#include "list_bits.hh"
namespace cList
{

template <typename T, typename Allocator =
std::allocator<list_bits::ListNode<T>>>
class CircleList {
   private:
    using Node = list_bits::ListNode<T>;
    size_t size_;
    [[no_unique_address]]
    Allocator allocator_{};
    Node* ringPtr_;

   public:
```

```
    using alocator_type = Allocator;
    using value_type = T;
    using reference = CircleList&;
    using pointer = CircleList*;
    using allocTraits_ = std::allocator_traits<Allocator>;
    using iterator = list_bits::Iterator<T>;
    using const_iterator = list_bits::Iterator<const T>;



    friend std::ostream& operator<<(std::ostream& os, const CircleList& obj)
{
        os << '[';
        for (size_t i = 0; i < obj.size_; i++) {
            os << obj[i] << " ";
        }
        os << ']';
        return os;
    }

    friend auto operator<=>(const CircleList& lhs, const CircleList& rhs) {
return lhs.size_ <=> rhs.size_; }
    friend bool operator==(const CircleList& lhs, const CircleList& rhs) {
        if (lhs.size_ != rhs.size_) return false;
        for (size_t i = 0; i < lhs.size_; i++) {
            if (lhs[i] != rhs[i]) return false;
        }
        return true;
    }

   public:
    CircleList() : size_{0}, ringPtr_{nullptr} {}

    explicit CircleList(size_t size) : size_{size}, ringPtr_{nullptr} {
        if (size_ == 0) return;
        ringPtr_ = allocator_.allocate(1);
        allocTraits_::construct(allocator_, ringPtr_);
        Node* current = ringPtr_;
        for (size_t i = 1; i < size_; ++i) {
            Node* newNode = allocator_.allocate(1);
            allocTraits_::construct(allocator_, newNode);
            current->next_ = newNode;
            newNode->prev_ = current;
            current = newNode;
        }
    }
    ~CircleList() { erase(); };
    CircleList(CircleList& other) : size_{other.size_},
ringPtr_{allocator_.allocate(1)} {
        allocTraits_::construct(allocator_, ringPtr_, other[0]);
        if (size_ == 1) {
            ringPtr_->next_ = ringPtr_;
            ringPtr_->prev_ = ringPtr_;
        }
        Node* current = ringPtr_;
```

```
        for (size_t i = 1; i < size_; ++i) {
            Node* newNode = allocator_.allocate(1);
            allocTraits_::construct(allocator_, newNode, other[i]);
            current->next_ = newNode;
            newNode->prev_ = current;
            current = newNode;
        }
        ringPtr_->prev_ = current;
        current->next_ = ringPtr_;
    }
    CircleList(CircleList&& other) noexcept : size_{other.size_},
ringPtr_{other.ringPtr_} { other.ringPtr_ = nullptr; }

    CircleList& operator=(CircleList& other) {
        *this = CircleList(other);
        return *this;
    }
    CircleList& operator=(CircleList&& other) noexcept {
        size_ = other.size_;
        allocator_ = std::move(other.allocator_);
        ringPtr_ = other.ringPtr_;
        other.ringPtr_ = nullptr;
        return *this;
    }

    bool isEmpty() const { return size_ == 0; }
    size_t size() const { return size_; }
    CircleList& swap(CircleList& other) noexcept {
        CircleList tmp{std::move(other)};
        other = std::move(*this);
        *this = std::move(tmp);
        return *this;
    }
    void pushFront(T& data) {
        pushBack(data);
        if (size_ != 1) {
            ringPtr_ = ringPtr_->prev_;
        }
    }
    void pushBack(T& data) {
        Node* tmpPtr = allocator_.allocate(1);
        allocTraits_::construct(allocator_, tmpPtr, data);
        if (size_ != 0) {
            tmpPtr->next_ = ringPtr_;
            tmpPtr->prev_ = ringPtr_->prev_;
            ringPtr_->prev_->next_ = tmpPtr;
            ringPtr_->prev_ = tmpPtr;
        } else {
            ringPtr_ = tmpPtr;
            ringPtr_->next_ = ringPtr_;
            ringPtr_->prev_ = ringPtr_;
        }
        size_++;
    }
    void pushFront(T&& data) {
```

```
        pushBack(std::move(data));
        if (size_ != 1) {
            ringPtr_ = ringPtr_->prev_;
        }
    }

    void pushBack(T&& data) {
        Node* tmpPtr = allocator_.allocate(1);
        allocTraits_::construct(allocator_, tmpPtr, std::move(data));
        if (size_ != 0) {
            tmpPtr->next_ = ringPtr_;
            tmpPtr->prev_ = ringPtr_->prev_;
            ringPtr_->prev_->next_ = tmpPtr;
            ringPtr_->prev_ = tmpPtr;
        } else {
            ringPtr_ = tmpPtr;
            ringPtr_->next_ = ringPtr_;
            ringPtr_->prev_ = ringPtr_;
        }
        size_++;
    }

    T popFront() {
        ringPtr_ = ringPtr_->next_;
        return popBack();
    }
    T popBack() {
        if (size_ == 0) throw std::length_error("Circle list is empty");
        Node* tmpPtr = ringPtr_->prev_;

        ringPtr_->prev_ = tmpPtr->prev_;
        tmpPtr->prev_->next_ = ringPtr_;

        T data = tmpPtr->data_;
        allocTraits_::destroy(allocator_, tmpPtr);
        return data;
    }

    T& operator[](const size_t index) const {
        Node* tmp = ringPtr_;
        for (size_t i = 0, count = index % size_; i < count; i++) {
            tmp = tmp->next_;
        }
        return tmp->data_;
    }
    const T& operator[](const size_t index, const int) const {
        Node* tmp = ringPtr_;
        for (size_t i = 0, count = index % size_; i < count; i++) {
            tmp = tmp->next_;
        }
        return tmp->data_;
    }
    void insert(const size_t index, T& data) {
        if (index >= size_) throw std::invalid_argument("Index out of
range");
```

```
        this->operator[](index) = data;
    }
    void insert(const size_t index, T&& data) {
        this->operator[](index) = std::move(data);
    }
    T& at(size_t index){
        if (index >= size_) throw std::invalid_argument("Index out of
range");
        return this->operator[](index);
    }
    const T& at(size_t index, int){
        if (index >= size_) throw std::invalid_argument("Index out of
range");
        return this->operator[](index);
    }

    iterator begin() { return iterator(ringPtr_); }
    iterator end() { return iterator(ringPtr_, ringPtr_, 1); }

    void erase() {
        if (!ringPtr_) return;
        Node* current = ringPtr_;
        for (size_t i = 0; i < size_; i++) {
            Node* tmp = current->next_;
            allocTraits_::destroy(allocator_, current);
            allocator_.deallocate(current, 1);
            current = tmp;
        }
        size_ = 0;
    };
};
} // namespace cList
```

Файл screens.cc:

```
#include <consoleUtils.hh>
#include <l8/include/list.hh>
#include <l8/include/list_algorithm.hh>
#include <print>

using namespace std;

using namespace console_utils;

namespace screens {
void printMainScreen() {
    auto [cols, rows] = getConsoleDimensions();
    println("{:^{}}", "\x{1B}[48;5;35mLab 8\x{1B}[0m", cols);
    println("Please select action:\n");
    println("    1.Input List");
    println("    2.Add element");
    println("    3.Clear List");
    println("    4.Sort List");
```

```
    println("    5.Find element");
    println("    6.Print list");
    println("    7.Exit");
}
bool inputList(cList::CircleList<double> &list) {
    list.erase();
    size_t num;
    readT(num, "Plese enter list length:", [](size_t number) { return number
> 0; }, "Number should be > 0");
    for (size_t i = 0; i < num; i++) {
        double data;
        readT(data, "Plese enter list element:");
        list.pushBack(data);
    }
    return true;
}
bool addElement(cList::CircleList<double> &list) {
    double data;
    readT(data, "Plese enter list element:");
    list.pushBack(data);
    return true;
}
bool clearList(cList::CircleList<double> &list) {
    list.erase();
    return true;
}
bool sortList(cList::CircleList<double> &list) {
    cList::bubbleSort(list.begin(), list.end());
    return true;
}
bool findElement(cList::CircleList<double> &list) {
    double data;
    readT(data, "Plese enter list element:");

    if (auto res = cList::find(list.begin(), list.end(), data); res ==
list.end()) {
        std::cout << "No such element\n";
    } else {
        std::cout << "List contains element\n";
    }
    return true;
}
bool printList(const cList::CircleList<double> &list) {
    std::cout << list << '\n';

    return true;
}
}  // namespace screens
```

Файл main.cc:

```
#include <consoleUtils.hh>
#include <iostream>
```

```
#include <l8/include/list.hh>
#include <l8/include/list_algorithm.hh>
#include <l8/include/screens.hh>
#include <functional>

using namespace std;
using namespace screens;
using namespace console_utils;
int main(void) {

    cList::CircleList<double> list;

    static array<function<bool()>, 7> actions = {
        [&list]() { return inputList(list); },
        [&list]() { return addElement(list); },
        [&list]() { return clearList(list); },
        [&list]() { return sortList(list); },
        [&list]() { return findElement(list); },
        [&list]() { return printList(list); },

        []() { return false; },

    };
    unsigned int response;
    do {
        printMainScreen();
        readT(response, ">", [](unsigned int num) { return num > 0 && num <=
7; });
        cout << "\x{1B}[2J\x{1B}[H\n";
    } while (actions[response - 1]());


    return 0;
}
```

Файл consoleUtils.hh:

```
#pragma once
#include <functional>
#include <iostream>
#include <limits>
#include <iomanip>
#include <chrono>
namespace console_utils {
std::pair<int, int> getConsoleDimensions();


template <typename T, typename CT>
void readT(T& data, const std::string& massage, CT validator) {
    std::cout << massage;
    while (((std::cin >> data).fail()) || !validator(data)) {
        std::cout << "Invalid input. Reread input requierments\n";
        std::cin.clear();
```

```
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        std::cout << massage;
    }
    std::cin.clear();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
template <typename T>
void readT(T& data, const std::string& massage) {
    std::cout << massage;
    while ((std::cin >> data).fail()) {
        std::cout << "Invalid input. Reread input requierments\n";
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        std::cout << massage;
    }
    std::cin.clear();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
template <typename T, typename CT>
void readT(T& data, const std::string& message, CT validator, const
std::string& errmess) {
    std::cout << message;
    while (((std::cin >> data).fail()) || !validator(data)) {
        std::cout << errmess;
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        std::cout << message;
    }
    std::cin.clear();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
template <typename T = std::chrono::sys_seconds, typename CT = const char * >
void readT(std::chrono::sys_seconds &data,  const char * massage, const char
* format) {
    std::cout << massage;
    while ((std::cin >> std::chrono::parse(format, data)).fail()) {
        std::cout << "Invalid input. Reread input requierments\n";
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        std::cout << massage;
    }
    std::cin.clear();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
}  // namespace console_utils
```

Файл consoleUtils.cc:

```
#include <iostream>

#ifdef __linux__
#include <sys/ioctl.h>
#include <unistd.h>
```

```
#endif

#ifdef _WIN32
#include <Windows.h>
#endif

namespace console_utils {
std::pair<int, int> getConsoleDimensions() {
#ifdef _WIN32
    CONSOLE_SCREEN_BUFFER_INFO csbi;
    GetConsoleScreenBufferInfo(GetStdHandle(STD_OUTPUT_HANDLE), &csbi);
    return std::make_pair(csbi.srWindow.Right - csbi.srWindow.Left + 1,
csbi.srWindow.Bottom - csbi.srWindow.Top + 1);
#endif
#ifdef __linux__

    struct winsize w;
    ioctl(STDOUT_FILENO, TIOCGWINSZ, &w);
    return std::make_pair(w.ws_col, w.ws_row);

#endif
}
}  // namespace console_utils
```

4 РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ

Рисунок 4.1 – Главное меню

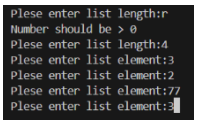

Рисунок 4.2 – Ввод кольца

Рисунок 4.3 – Вывод кольца

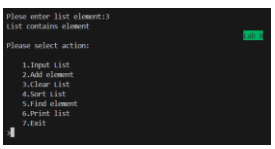

Рисунок 4.4 – Поиск элемента

Рисунок 4.4 – Сортировка

5 ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы был успешно реализован класс контейнера «двунаправленное кольцо» с поддержкой итераторов, а также алгоритмы поиска и сортировки. Реализация соответствует принципам STL: обеспечена корректная работа с итераторами, поддержка семантики перемещения, использование аллокаторов и совместимость с стандартными алгоритмами.