

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«ПЕРМСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

ЭЛЕКТРОТЕХНИЧЕСКИЙ ФАКУЛЬТЕТ

КАФЕДРА «ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И  
АВТОМАТИЗИРОВАННЫХ СИСТЕМ»

**ОТЧЁТ**  
**«ЛАБОРАТОРНАЯ №4: ЦИКЛИЧЕСКИЙ ДВУСВЯЗНЫЙ СПИСОК**  
**МЕТОДЫ СОРТИРОВКИ: ШЕЛЛА, ХОАРА»**

Дисциплина: «Программирование»

Выполнил:

Студент группы ИВТ-21-26

Безух Владимир Сергеевич

Проверил:

Доцент кафедры ИТАС

Полякова Ольга Андреевна

Пермь, 2022

## Содержание

1. Постановка задач .....	3
2. Анализ задач .....	4
3. Описание структуры Circular Doubly Linked List.....	8
4. Исходный код .....	10
5. Анализ результатов .....	23

## **1. Постановка задач**

а) Реализовать структуру данных — список (list). Особенности предложенной реализации: циклический (кольцевой) двунаправленный список (Circular Doubly Linked List).

б) Отсортировать произвольный список с помощью разных методов сортировки: Шелла, Хоара (quick sort).

## 2. Анализ задач

а) Классический массив и список отличаются способом организации работы с памятью для хранения данных.

Данные в массиве хранятся в непрерывном участке памяти, данные в списке хранятся в произвольных участках памяти.

Доступ к элементам массива осуществляется с помощью указателя на первый элемент массива через адресную арифметику, доступ к элементам списка осуществляется через указатели на каждый конкретный элемент.

Хранение массива указателей на элементы накладывает свойственные массиву ограничения, поэтому такой подход не имеет отношения к спискам. Список состоит из узлов (рис. 1) — совокупности непосредственно хранимых данных и произвольного количества указателей на другие узлы («следующий», «предыдущий», произвольные «опорные» и т. д.) списка (зависит от реализации).

Списки классифицируются по двум основным признакам: количеству указателей в узле (односвязные, двусвязные, многосвязные) и цикличностью (ссылается ли «последний» элемент списка на «первый элемент» или нет). В циклическом списке обычно выделяют произвольный «начальный» узел», в «обычном» списке — «головной» и «хвостовой» узлы.

Из вышеописанных ключевых различий вытекают особенности применения массива и списка. Нельзя сказать, что как структура данных массив/список «однозначно лучше списка/массива». Важно учитывать область применения, использовать «предпочтительные инструменты» в ходе решения поставленной задачи.



Рисунок 1 — Схематичное представление циклического двусвязного списка

Основные особенности массива и списка.

В списке проще организовать хранение данных разных размеров. Все данные в массиве имеют одинаковый размер.

Доступ к данным в массиве осуществляется «напрямую» за константное время. В списке необходимо переходить от одного элемента к другому за линейное время.

Для изменения количества узлов в списке достаточно выделить/высвободить память для конкретного узла и переуказать указатели. Для изменения количества элементов в массиве необходимо выделить новый участок памяти требуемого размера, перенести элементы в новый массив, удалить предыдущий массив. Любое изменение размера массива «затратное» по памяти и производительности.

Массивы не подходят для распараллеливания процессов. Многократные изменения размеров массивов (разделение/слияние) «серьёзно» сказываются на затратах памяти и производительности. Списки разделяются/сливаются через переуказания указателей, что «намного быстрее» и не требует выделения дополнительной памяти.

Массивы не подходят для многопоточных процессов. Одновременное изменение массива разными потоками приводит к непредсказуемым ошибкам. Добавлять/удалять элементы в массиве нужно последовательно, потому что каждый раз происходят промежуточные операции выделения/освобождения памяти. Например, если два разных потока одновременно попытаются изменить массив, получится две изменённые копии исходного массива. Что делать в такой ситуации — не ясно. Списки подходят для многопоточных процессов, т.к. при согласованном совместном изменении разных частей списка разными потоками, каждый участник процесса работает со своими «независимыми» узлами.

Разные списки могут работать с разной интерпретацией одних и тех же данных (участков памяти). Участки памяти массивов не пересекаются.

## б) Сортировка Шелла (рис. 2).

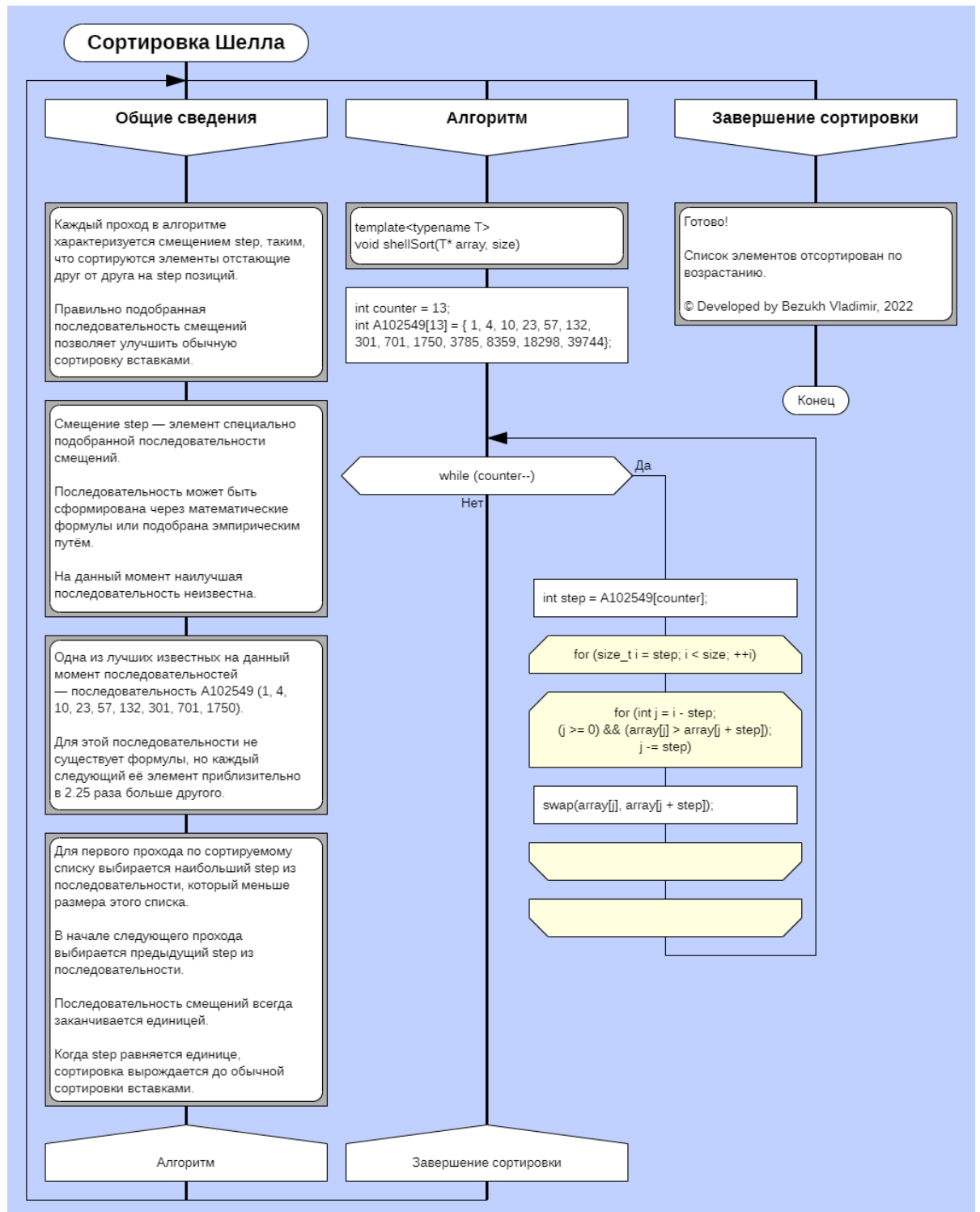


Рисунок 2 — Алгоритм сортировки методом Шелла на языке ДРАКОН

## в) Сортировка Хоара (рис. 3).

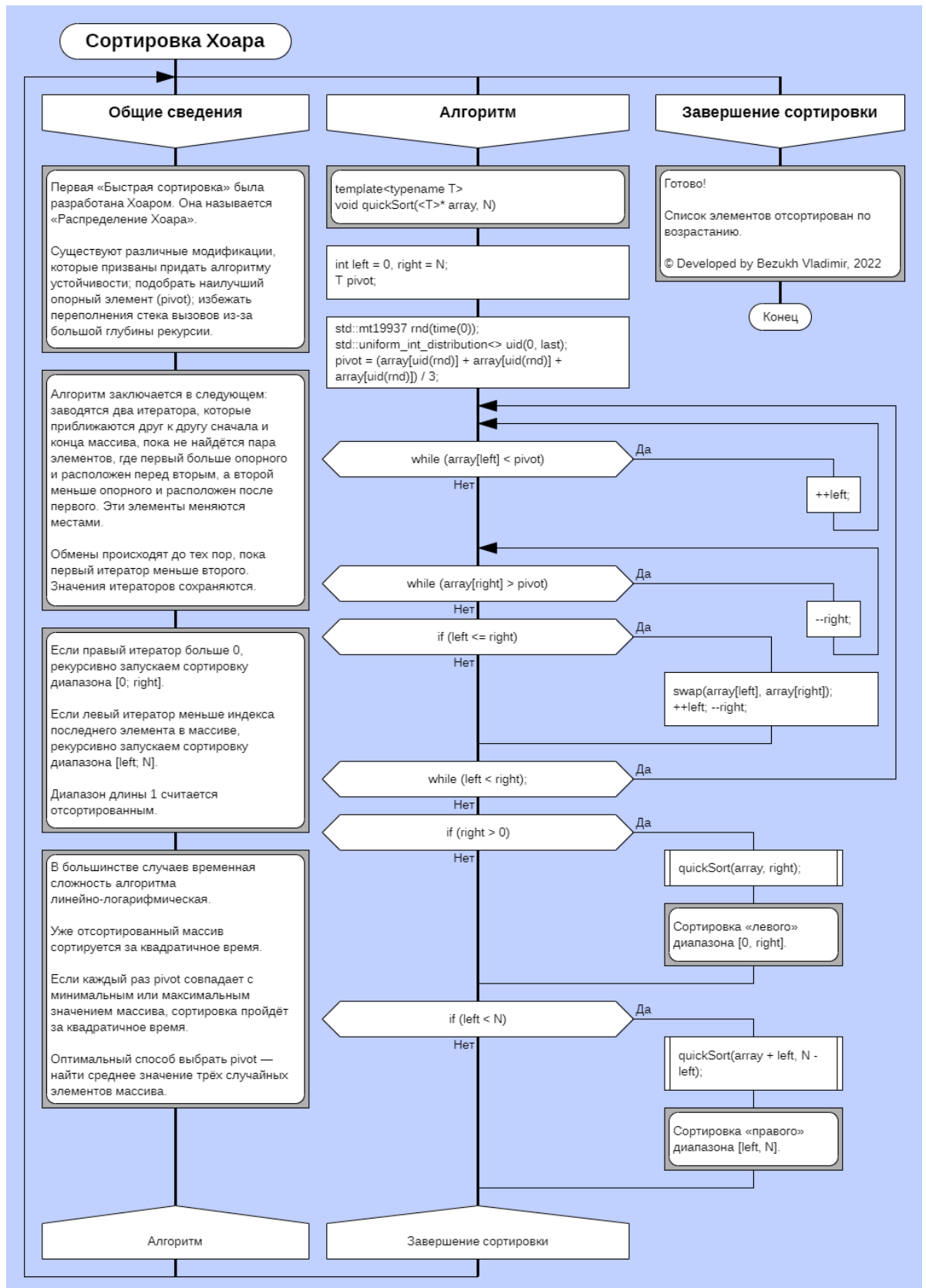


Рисунок 3 — Алгоритм сортировки методом Хоара на языке ДРАКОН

### 3. Описание структуры Circular Doubly Linked List

В классе CircularDoublyLinkedList находятся поля и функции-члены для работы со списком. Вложенный класс Node, доступный только в области видимости экземпляров класса CircularDoublyLinkedList, содержит поля для хранения данных и указателей на «следующий» и «предыдущий» узлы. Оба класса шаблонные, что позволяет работать с произвольными типами данных.

```
template<typename T>
class CircularDoublyLinkedList
{
public:
    CircularDoublyLinkedList();
    ~CircularDoublyLinkedList();

    // Здесь находятся прототипы функций-членов, для удобства описанных вне класса

    // Здесь находятся прототипы перегруженных операторов

private:
    template<typename T>
    class Node
    {
    public:
        Node* pointer_to_prev_node;
        Node* pointer_to_next_node;
        T data;

        // Здесь находится конструктор, функция-член и перегрузка оператора для класса Node
    };

    // Здесь находятся прототипы частных функций-членов, недоступных извне

    int list_size; // Размер списка
    Node<T>* start_node; // Указатель на начальный узел.

    // Указатель на «текущий» узел и его индекс.
    // В том числе означает длину кратчайшего пути от начального узла к текущему узлу.
    // Если индекс меньше нуля, то кратчайший путь против часовой стрелки, и наоборот.
    int current_node_shift;
    Node<T>* current_node;
};
```



Описание некоторых функций-членов:

```
size_t getSize() { return this->list_size; }
```

Геттер для инкапсуляции размера списка.

```
void changeStartNode(const int index);
```

Можно изменить указатель на начальный узел.

```
void resetCurrentNode();
```

Сброс текущего узла на начальный. Например, можно применять при изменении размера списка, чтобы не пересчитывать хранимый индекс текущего указателя.

```
void insertToLeft(const T& data, const int index = 0);  
void insertToRight(const T& data, const int index = 0);
```

Добавление нового узла в список слева или справа. Если список пуст, создаётся начальный узел. На вход принимается два параметра: данные и индекс узла, слева или справа от которого нужно вставить новый узел. По умолчанию второй параметр равен нулю и его можно не указывать.

```
void swapTwoNodes(int first, int second);
```

Перестановка двух узлов «местами». На самом деле данные остаются в тех же участках памяти — лишь указатели переуказываются в соответствии с новым расположением узлов в списке. Менять местами непосредственно данные узлов можно с помощью перегруженного оператора []. Работа с указателями приоритетнее — перестановка двух элементов местами, особенно если это сложные пользовательские типы данных, «куда менее эффективна» по памяти и производительности.

```
void clear();
```

Удаление всех узлов из списка.

```
void removeToLeft(const int index = 0);  
void removeToRight(const int index = 0);
```

Удаление узла слева или справа от элемента по индексу. Параметр по умолчанию можно не указывать.

## 4. Исходный код

```
#include <random>
#include <chrono>
#include <cassert>
#include <iostream>
#include <vld.h>

class Timer
{
public:
    void start() {
        this->s = std::chrono::high_resolution_clock::now();
    }

    long long finish() {
        this->f = std::chrono::high_resolution_clock::now();
        return std::chrono::duration_cast<std::chrono::milliseconds>(this->f - this->s).count();
    }

private:
    std::chrono::time_point<std::chrono::steady_clock> s, f;
};

template<typename T>
class CircularDoublyLinkedList
{
public:
    CircularDoublyLinkedList();
    ~CircularDoublyLinkedList();

    int getSize() { return this->list_size; }

    void changeStartNode(const int index);
    void resetCurrentNode();

    void insertToLeft(const T& data, const int index = 0);
    void insertToRight(const T& data, const int index = 0);

    void swapTwoNodes(int first, int second);

    void clear();
    void reset();
    void removeToLeft(const int index = 0);
    void removeToRight(const int index = 0);

    CircularDoublyLinkedList<T>* split_MemoryForNewListIsAllocatedThroughNew(const int& split_index);
    void merge(CircularDoublyLinkedList<T>* another_list);

    void mergeSort();
    void shellSort();

    T& operator[](const int& index);
    CircularDoublyLinkedList<T>& operator=(const CircularDoublyLinkedList<T>& right);
};
```

```

private:
    template<typename T>
    class Node
    {
    public:
        Node* pointer_to_prev_node;
        Node* pointer_to_next_node;
        T data;

        Node(T data = T(), Node* pointer_to_prev_node = nullptr, Node* pointer_to_next_node
= nullptr) {
            this->data = data;
            this->pointer_to_prev_node = pointer_to_prev_node;
            this->pointer_to_next_node = pointer_to_next_node;
        }

        // Changes the position of two nodes (this and another) in the list by changing the
pointers
        void replacePointers(Node<T>* another) {
            bool flag_to_prev_of_this_node_is_regular_node = this->pointer_to_prev_node !=
another;
            bool flag_to_next_of_this_node_is_regular_node = this->pointer_to_next_node !=
another;

            bool flag_list_size_is_not_equal_to_two =
flag_to_prev_of_this_node_is_regular_node || flag_to_next_of_this_node_is_regular_node;
            assert(flag_list_size_is_not_equal_to_two && "Use changeStartNode() method");

            // Demonstration of node positions
            if (flag_to_prev_of_this_node_is_regular_node) // // regular - this - ?
                this->pointer_to_prev_node->pointer_to_next_node = another;
            else { // regular - other - this - regular
                another->pointer_to_prev_node->pointer_to_next_node = this;
                this->pointer_to_next_node->pointer_to_prev_node = another;

                this->pointer_to_prev_node = another->pointer_to_prev_node;
                another->pointer_to_next_node = this->pointer_to_next_node;

                this->pointer_to_next_node = another;
                another->pointer_to_prev_node = this;
                return;
            }

            if (flag_to_next_of_this_node_is_regular_node) // regular - this - regular
                this->pointer_to_next_node->pointer_to_prev_node = another;
            else { // regular - this - other - regular
                this->pointer_to_next_node = another->pointer_to_next_node;
                this->pointer_to_next_node->pointer_to_prev_node = this;

                another->pointer_to_prev_node = this->pointer_to_prev_node;

                this->pointer_to_prev_node = another;
                another->pointer_to_next_node = this;
                return;
            }

            // ... other ... regular - this - regular ... other ...
            Node<T>* temp_prev = this->pointer_to_prev_node;
            Node<T>* temp_next = this->pointer_to_next_node;

            this->pointer_to_prev_node = another->pointer_to_prev_node;
            this->pointer_to_next_node = another->pointer_to_next_node;
            this->pointer_to_prev_node->pointer_to_next_node = this;
            this->pointer_to_next_node->pointer_to_prev_node = this;

```

```

        another->pointer_to_prev_node = temp_prev;
        another->pointer_to_next_node = temp_next;
    }

    Node<T>& operator=(const Node<T>& right) {
        if (this != &right) {
            this->data = right.data;
            this->pointer_to_prev_node = right.pointer_to_prev_node;
            this->pointer_to_next_node = right.pointer_to_next_node;
        }

        return *this;
    }
};

void setCurrentNode(int index);

int list_size;
Node<T>* start_node;

int current_node_shift; // Current node index. Shortest way to current node: distance
and direction
Node<T>* current_node; // Node to optimize iterations. To find the needful node, use
setCurrentNode() method
};

template<typename T>
CircularDoublyLinkedList<T>::CircularDoublyLinkedList()
{
    this->list_size = 0;
    this->start_node = nullptr;

    this->current_node_shift = 0;
    this->current_node = nullptr;
}

template<typename T>
CircularDoublyLinkedList<T>::~CircularDoublyLinkedList()
{ clear(); }

template<typename T>
void CircularDoublyLinkedList<T>::changeStartNode(const int index)
{
    assert(this->list_size && "List must not be empty");

    if (index % this->list_size) { // if index is not 0
        setCurrentNode(index);

        this->start_node = current_node;
        this->resetCurrentNode();
    }
}

template<typename T>
void CircularDoublyLinkedList<T>::resetCurrentNode()
{
    this->current_node = this->start_node;
    this->current_node_shift = 0;
}

```

```

template<typename T>
void CircularDoublyLinkedList<T>::insertToLeft(const T& data, const int index)
{
    Node<T>* insert = new Node<T>(data);

    if (this->list_size) {
        setCurrentNode(index);
        Node<T>* current = this->current_node;

        insert->pointer_to_prev_node = current->pointer_to_prev_node;
        insert->pointer_to_next_node = current;

        current->pointer_to_prev_node->pointer_to_next_node = insert;
        current->pointer_to_prev_node = insert;

        this->resetCurrentNode();
    } else {
        insert->pointer_to_prev_node = insert;
        insert->pointer_to_next_node = insert;

        this->start_node = insert;
        this->current_node = insert;
    }

    ++list_size;
}

template<typename T>
void CircularDoublyLinkedList<T>::insertToRight(const T& data, const int index)
{
    Node<T>* insert = new Node<T>(data);

    if (this->list_size) {
        setCurrentNode(index);
        Node<T>* current = this->current_node;

        insert->pointer_to_prev_node = current;
        insert->pointer_to_next_node = current->pointer_to_next_node;

        current->pointer_to_next_node->pointer_to_prev_node = insert;
        current->pointer_to_next_node = insert;

        this->resetCurrentNode();
    } else {
        insert->pointer_to_prev_node = insert;
        insert->pointer_to_next_node = insert;

        this->start_node = insert;
        this->current_node = insert;
    }

    ++list_size;
}

```

```

template<typename T>
void CircularDoublyLinkedList<T>::swapTwoNodes(int first, int second)
{
    assert(this->list_size > 1 && "List size must be greater than 1");

    // Indexes normalization
    first %= this->list_size; if (first < 0) first += this->list_size;
    second %= this->list_size; if (second < 0) second += this->list_size;

    if (first != second) { // It makes no sense to swap the same node
        this->setCurrentNode(first);
        Node<T>* first_node = this->current_node;

        this->setCurrentNode(second);
        Node<T>* second_node = this->current_node;

        if (this->list_size > 2) // If the list size is two, it is sufficient to change the
start node
            first_node->replacePointers(second_node);

        // If one of the nodes is a start node
        if (first_node == this->start_node)
            this->start_node = second_node;
        else if (second_node == this->start_node)
            this->start_node = first_node;

        this->resetCurrentNode();
    }
}

template<typename T>
void CircularDoublyLinkedList<T>::clear()
{
    Node<T>* remove;
    Node<T>* next_remove = this->start_node;

    while (this->list_size)
    {
        remove = next_remove;
        next_remove = remove->pointer_to_next_node;

        delete remove;
        --list_size;
    }

    reset();
}

template<typename T>
void CircularDoublyLinkedList<T>::reset()
{
    this->list_size = 0;
    this->start_node = nullptr;

    this->current_node_shift = 0;
    this->current_node = nullptr;
}

```

```

template<typename T>
void CircularDoublyLinkedList<T>::removeToLeft(const int index)
{
    assert(this->list_size && "List must not be empty");

    setCurrentNode(index);
    Node<T>* current = this->current_node;
    Node<T>* remove = current->pointer_to_prev_node;

    current->pointer_to_prev_node = remove->pointer_to_prev_node;
    remove->pointer_to_prev_node->pointer_to_next_node = current;

    if (remove == this->start_node)
        this->start_node = current;

    this->resetCurrentNode();
    delete remove;
    --list_size;
}

template<typename T>
void CircularDoublyLinkedList<T>::removeToRight(const int index)
{
    assert(this->list_size && "List must not be empty");

    setCurrentNode(index);
    Node<T>* current = this->current_node;
    Node<T>* remove = current->pointer_to_next_node;

    current->pointer_to_next_node = remove->pointer_to_next_node;
    remove->pointer_to_next_node->pointer_to_prev_node = current;

    if (remove == this->start_node)
        this->start_node = current;

    this->resetCurrentNode();
    delete remove;
    --list_size;
}

template<typename T>
CircularDoublyLinkedList<T>*
CircularDoublyLinkedList<T>::split_MemoryForNewListIsAllocatedThroughNew(const int&
split_index) {
    assert(this->list_size > 1 && "List must be greater than 1");
    assert(split_index % this->list_size && "Can't divide this list at its start node");

    CircularDoublyLinkedList<T>* new_list = new CircularDoublyLinkedList<T>;

    this->setCurrentNode(split_index - 1);
    new_list->start_node = this->current_node->pointer_to_next_node;
    new_list->list_size = this->list_size - split_index;
    new_list->resetCurrentNode();

    this->current_node->pointer_to_next_node = this->start_node;
    new_list->start_node->pointer_to_prev_node = this->start_node->pointer_to_prev_node;
    this->start_node->pointer_to_prev_node = this->current_node;
    new_list->start_node->pointer_to_prev_node->pointer_to_next_node = new_list->start_node;

    this->list_size = split_index;
    this->resetCurrentNode();

    return new_list;
}

```

```

template<typename T>
void CircularDoublyLinkedList<T>::merge(CircularDoublyLinkedList<T>* another_list)
{
    if (this->list_size == 0) {
        *this = *another_list;
        another_list->reset();
        return;
    }

    if (another_list->list_size == 0)
        return;

    Node<T>* this_list_tail_node = this->start_node->pointer_to_prev_node;
    Node<T>* another_list_tail_node = another_list->start_node->pointer_to_prev_node;

    this_list_tail_node->pointer_to_next_node = another_list->start_node;
    another_list_tail_node->pointer_to_next_node = this->start_node;

    this->start_node->pointer_to_prev_node = another_list_tail_node;
    another_list->start_node->pointer_to_prev_node = this_list_tail_node;

    this->list_size += another_list->list_size;
    this->resetCurrentNode();

    another_list->reset();
}

template<typename T>
void CircularDoublyLinkedList<T>::shellSort()
{
    int counter = 13;
    int A102549[13] = { 1, 4, 10, 23, 57, 132, 301, 701, 1750, 3785, 8359, 18298, 39744 };
    // https://oeis.org/A102549

    while (counter--) {
        int step = A102549[counter];

        for (int i = step; i < this->list_size; ++i)
            for (int j = i - step; (j >= 0) && ((*this)[j] > (*this)[j + step]); j -= step)
                this->swapTwoNodes(j, j + step);
    }
}

```



```

template<typename T>
void CircularDoublyLinkedList<T>::mergeSort()
{
    size_t block_size_iterator, block_iterator,
        left_block_iterator, right_block_iterator, merge_iterator,
        left_border, middle, right_border;

    for (block_size_iterator = 1; block_size_iterator < this->list_size; block_size_iterator
<<= 1)
    {
        for (block_iterator = 0;
            block_iterator < (this->list_size - block_size_iterator);
            block_iterator += (block_size_iterator << 1))
        {
            left_block_iterator = 0; right_block_iterator = 0;
            left_border = block_iterator; middle = block_iterator + block_size_iterator;
            right_border = block_iterator + (block_size_iterator << 1);

            if (right_border > this->list_size)
                right_border = this->list_size;

            T* sorted_block = new T[right_border - left_border];

            while ((left_border + left_block_iterator) < middle && (middle +
right_block_iterator) < right_border)
            {
                T temp_left = (*this)[left_border + left_block_iterator];
                T temp_right = (*this)[middle + right_block_iterator];

                if (temp_left < temp_right)
                {
                    sorted_block[left_block_iterator + right_block_iterator] = temp_left;
                    ++left_block_iterator;
                }
                else
                {
                    sorted_block[left_block_iterator + right_block_iterator] = temp_right;
                    ++right_block_iterator;
                }
            }

            while (left_border + left_block_iterator < middle)
            {
                T temp_left = (*this)[left_border + left_block_iterator];
                sorted_block[left_block_iterator + right_block_iterator] = temp_left;
                ++left_block_iterator;
            }

            while (middle + right_block_iterator < right_border)
            {
                T temp_right = (*this)[middle + right_block_iterator];
                sorted_block[left_block_iterator + right_block_iterator] = temp_right;
                ++right_block_iterator;
            }

            for (merge_iterator = 0; merge_iterator < (left_block_iterator +
right_block_iterator); ++merge_iterator)
                (*this)[left_border + merge_iterator] = sorted_block[merge_iterator];

            delete [] sorted_block;
        }
    }
}

```

```

template<typename T>
T& CircularDoublyLinkedList<T>::operator[](const int& index)
{
    assert(this->list_size && "List must not be empty");

    setCurrentNode(index);
    return this->current_node->data;
}

template<typename T>
CircularDoublyLinkedList<T>& CircularDoublyLinkedList<T>::operator=(const
CircularDoublyLinkedList<T>& right)
{
    if (this != &right) {
        this->list_size = right.list_size;
        this->start_node = right.start_node;
        this->current_node = right.current_node;
        this->current_node_shift = right.current_node_shift;
    }

    return *this;
}

template<typename T>
void CircularDoublyLinkedList<T>::setCurrentNode(int index)
{
    assert(this->list_size && "List must not be empty");

    index %= this->list_size; // -(list_size - 1) ... 0 ... (list_size - 1)

    if (index == 0) {
        this->current_node_shift = 0;
        this->current_node = this->start_node;
        return;
    }
    else if (index < 0) index += this->list_size;

    size_t clockwise_distance_from_start_node,
        counterclockwise_distance_from_start_node,
        clockwise_distance_from_current_node,
        counterclockwise_distance_from_current_node;

    clockwise_distance_from_start_node = index;
    counterclockwise_distance_from_start_node = this->list_size - index;

    int normalized_current_node_shift = this->current_node_shift;
    if (normalized_current_node_shift < 0) normalized_current_node_shift += this->list_size;

    if (normalized_current_node_shift > index) {
        clockwise_distance_from_current_node = (this->list_size -
normalized_current_node_shift) + clockwise_distance_from_start_node;
        counterclockwise_distance_from_current_node = normalized_current_node_shift - index;
    } else {
        clockwise_distance_from_current_node = index - normalized_current_node_shift;
        counterclockwise_distance_from_current_node = normalized_current_node_shift +
counterclockwise_distance_from_start_node;
    }

    bool flag_clockwise_from_start_node;
    bool flag_clockwise_from_current_node;

    size_t minimal_distance_from_start_node;
    if (clockwise_distance_from_start_node < counterclockwise_distance_from_start_node) {
        flag_clockwise_from_start_node = true;
        minimal_distance_from_start_node = clockwise_distance_from_start_node;
    }

```

```

    } else {
        flag_clockwise_from_start_node = false;
        minimal_distance_from_start_node = counterclockwise_distance_from_start_node;
    }

    size_t minimal_distance_from_current_node;
    if (clockwise_distance_from_current_node < counterclockwise_distance_from_current_node)
    {
        flag_clockwise_from_current_node = true;
        minimal_distance_from_current_node = clockwise_distance_from_current_node;
    } else {
        flag_clockwise_from_current_node = false;
        minimal_distance_from_current_node = counterclockwise_distance_from_current_node;
    }

    bool flag_clockwise;
    size_t minimal_distance;
    bool flag_start_node_selected = false;

    if (minimal_distance_from_start_node < minimal_distance_from_current_node) {
        flag_start_node_selected = true;
        flag_clockwise = flag_clockwise_from_start_node;
        minimal_distance = minimal_distance_from_start_node;
    } else {
        flag_start_node_selected = false;
        flag_clockwise = flag_clockwise_from_current_node;
        minimal_distance = minimal_distance_from_current_node;
    }

    Node<T>* current;
    if (flag_start_node_selected) {
        current = this->start_node;
    } else {
        current = this->current_node;
    }

    if (flag_clockwise) {
        for (size_t shift = 0; shift != minimal_distance; ++shift) {
            current = current->pointer_to_next_node;
        }

        this->current_node = current;

        if (flag_start_node_selected) {
            this->current_node_shift = minimal_distance;
        } else {
            if (this->current_node_shift < 0) {
                this->current_node_shift += this->list_size;
            }

            this->current_node_shift += minimal_distance;
        }
    } else {
        for (size_t shift = 0; shift != minimal_distance; ++shift) {
            current = current->pointer_to_prev_node;
        }

        this->current_node = current;

        if (flag_start_node_selected) {
            this->current_node_shift = -(int)minimal_distance;
        }
        else {
            if (this->current_node_shift > 0) {
                this->current_node_shift -= this->list_size;
            }
        }
    }

```

```

        }

        this->current_node_shift -= minimal_distance;
    }
}

// Finding best direction
if (this->current_node_shift > 0) {
    if (this->current_node_shift > this->list_size / 2)
        this->current_node_shift -= this->list_size;
    }
else {
    if (-(this->current_node_shift) > this->list_size / 2)
        this->current_node_shift += this->list_size;
    }
}

template<typename T>
CircularDoublyLinkedList<T>* quickSort(CircularDoublyLinkedList<T>* list)
{
    int last = list->getSize() - 1;

    if (list->getSize() == 1) return list;

    int left = 0, right = last;
    T pivot;

    std::mt19937 rnd(time(0));
    std::uniform_int_distribution<> uid(0, last);
    pivot = ((*list)[uid(rnd)] + (*list)[uid(rnd)] + (*list)[uid(rnd)]) / 3;

    do {
        while ((*list)[left] < pivot) ++left;
        while ((*list)[right] > pivot) --right;

        if (left <= right) {
            list->swapTwoNodes(left, right);
            ++left; --right;
        }
    } while (left < right);

    CircularDoublyLinkedList<T>* temp_left;
    CircularDoublyLinkedList<T>* temp_right;

    if (right > 0) {
        temp_right = list->split_MemoryForNewListIsAllocatedThroughNew(right + 1);
        temp_left = quickSort(list);
        temp_left->merge(temp_right);
        list = temp_left;

        temp_right->reset();
        delete temp_right;
    }

    if (left < last) {
        temp_right = list->split_MemoryForNewListIsAllocatedThroughNew(left);
        list->merge(quickSort(temp_right));

        temp_right->reset();
        delete temp_right;
    }

    return list;
}

```

```

void test()
{
    std::mt19937 rnd(time(0));
    std::uniform_int_distribution<> uid(100, 999);

    CircularDoublyLinkedList<int>* list1 = new CircularDoublyLinkedList<int>;
    CircularDoublyLinkedList<int>* list2 = new CircularDoublyLinkedList<int>;
    CircularDoublyLinkedList<int>* list3 = new CircularDoublyLinkedList<int>;
    size_t size; std::cout << "List size: "; std::cin >> size;

    for (size_t i = size; i != 0; --i) {
        int data = uid(rnd);
        list1->insertToLeft(data);
        list2->insertToLeft(data);
        list3->insertToLeft(data);
    }

    std::cout << "\n" << "Unsorted list:\n";
    for (size_t i = size; i != 0; --i) {
        if (i % 20 == 0)
            std::cout << "\n";
        std::cout << (*list1)[i] << " ";
    }

    Timer timer;
    long long time1, time2, time3;
    timer.start(); list1->mergeSort(); time1 = timer.finish();
    timer.start(); list2->shellSort(); time2 = timer.finish();
    timer.start(); list3->quickSort(list3); time3 = timer.finish();

    std::cout << "\n\n" << "Sorted first list (merge sort):\n";
    for (size_t i = 0; i != size; ++i) {
        if (i && ((i % 20) == 0))
            std::cout << "\n";
        std::cout << (*list1)[i] << " ";
    }

    std::cout << "\n\n" << "Sorted second list (shell sort):\n";
    for (size_t i = 0; i != size; ++i) {
        if (i && ((i % 20) == 0))
            std::cout << "\n";
        std::cout << (*list2)[i] << " ";
    }

    std::cout << "\n\n" << "Sorted third list (quick sort):\n";
    for (size_t i = 0; i != size; ++i) {
        if (i && ((i % 20) == 0))
            std::cout << "\n";
        std::cout << (*list3)[i] << " ";
    }

    bool flag_fail1 = false, flag_fail2 = false, flag_fail3 = false;
    std::cout << "\n\nChecking:";
    for (size_t i = 1; i != size; ++i)
    {
        flag_fail1 = (*list1)[i - 1] > (*list1)[i];
        flag_fail2 = (*list2)[i - 1] > (*list2)[i];
        flag_fail3 = (*list3)[i - 1] > (*list3)[i];
    }

    std::cout << "\nFirst list sorted (merge sort) in " << time1 << " ms: " << (flag_fail1 ?
"\tfailed!" : "\tsuccess!");
    std::cout << "\nSecond list sorted (shell sort) in " << time2 << " ms: " << (flag_fail2
? "\tfailed!" : "\tsuccess!");

```

```

        std::cout << "\nThird list sorted (quick sort) in " << time3 << " ms: " << (flag_fail3 ?
"\tfailed!" : "\tsuccess!");
        std::cout << "\n\n";

        std::cout << "Quick sort is " << (double)time1 / time3 << " times faster than merge
sort!\n";
        std::cout << "Quick sort is " << (double)time2 / time3 << " times faster than shell
sort!\n";
        std::cout << "Merge sort is " << (double)time2 / time1 << " times faster than shell
sort!\n\n";

        delete list1;
        delete list2;
        delete list3;
    }

    int main()
    {
        test();
        return 0;
    }

```

## 5. Анализ результатов

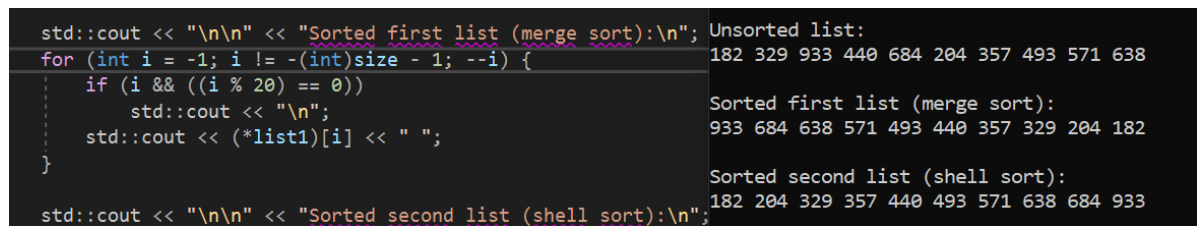
а) Структура много чего умеет. Например.

Добавлять и убирать новые узлы в произвольных местах.

Соединять списки воедино и делить в произвольных местах.

Можно менять начальный узел. Индексация ведётся в обе стороны, т.е. можно использовать отрицательные индексы. Можно использовать произвольные индексы — при обращении по индексу происходит взятие модуля по размеру списка. Например, если в списке 10 элементов, при обращении по индексу -42 произойдёт следующее:  $-42 \% \text{list\_size} \rightarrow -2$  или 8 индекс.

Поиск узла оптимизирован таким образом, что через простые расчёты с помощью номера индекса и размера массива проверяются всевозможные пути по часовой/против часовой стрелки от `start_node` и `current_node`. Таким образом, кратчайший путь при поиске узла не превышает  $\text{list\_size} / 2$  шагов. При поиске узла в `current_node` сохраняется последний найденный узел, поэтому при последовательной итерации по циклу каждый последующий узел находится за один шаг (рис 4.).



```
std::cout << "\n\n" << "Sorted first list (merge sort):\n";
for (int i = -1; i != -(int)size - 1; --i) {
    if (i && ((i % 20) == 0))
        std::cout << "\n";
    std::cout << (*list1)[i] << " ";
}
std::cout << "\n\n" << "Sorted second list (shell sort):\n";
```

Unsorted list:  
182 329 933 440 684 204 357 493 571 638

Sorted first list (merge sort):  
933 684 638 571 493 440 357 329 204 182

Sorted second list (shell sort):  
182 204 329 357 440 493 571 638 684 933

Рисунок 4 — Отсортированный список, выведенный в «обратном» порядке. Направление в списке — условность.

К данным в узлах можно обращаться через перегруженный оператор `[]`. Важно помнить, что перестановка «тяжёлых» пользовательских типов данных местами может вызвать «серьёзные» затраты производительности. Сами по себе узлы находятся в произвольных местах памяти, поэтому для смены двух значений в списке достаточно переуказать все соприкасающиеся указатели, чтобы именно узлы встали на новые позиции в иерархии списка. Такой подход предпочтительнее и реализован через метод `swapTwoNodes(int first, int second)`.

Стоит признать, что структура неидеальна и ещё есть над чем работать. Например, компилятор clang ругается на одинаковые названия шаблонных типов у внешнего и вложенного класса. Компилятор gcc не задаёт лишних вопросов и спокойно всё компилирует. На данный момент я не знаю, как грамотно отрефакторить свой код, чтобы решить эту проблему.

Работа со сложными структурами данных предполагает постоянные манипуляции с памятью и указателями. Деструктор списка устроен таким образом, что он полностью удаляет все связанные с ним узлы. Это значит, что нужно быть осторожным, если разные списки имеют общие узлы. Если мы не хотим, чтобы не нужный нам список при вызове деструктора уничтожил все сопутствующие узлы, необходимо осуществить сброс размера списка через функцию-член `reset()`. Особенно важно внимательно следить за памятью при разбиении и слиянии списков.

Например (рис. 5), при разбиении списка надвое новый был выделен динамически через `new`, поэтому его нужно удалить. Но сделать это нужно только после того, как были соединены отсортированные части. Чтобы не повредить общие узлы, перед удалением списка из памяти нужно сбросить его размер через функцию-член класса `reset()`.

```
if (right > 0) {
    temp_right = list->split_MemoryForNewListIsAllocatedThroughNew(right + 1);
    temp_left = quickSort(list);
    temp_left->merge(temp_right);
    list = temp_left;

    temp_right->reset();
    delete temp_right;
}

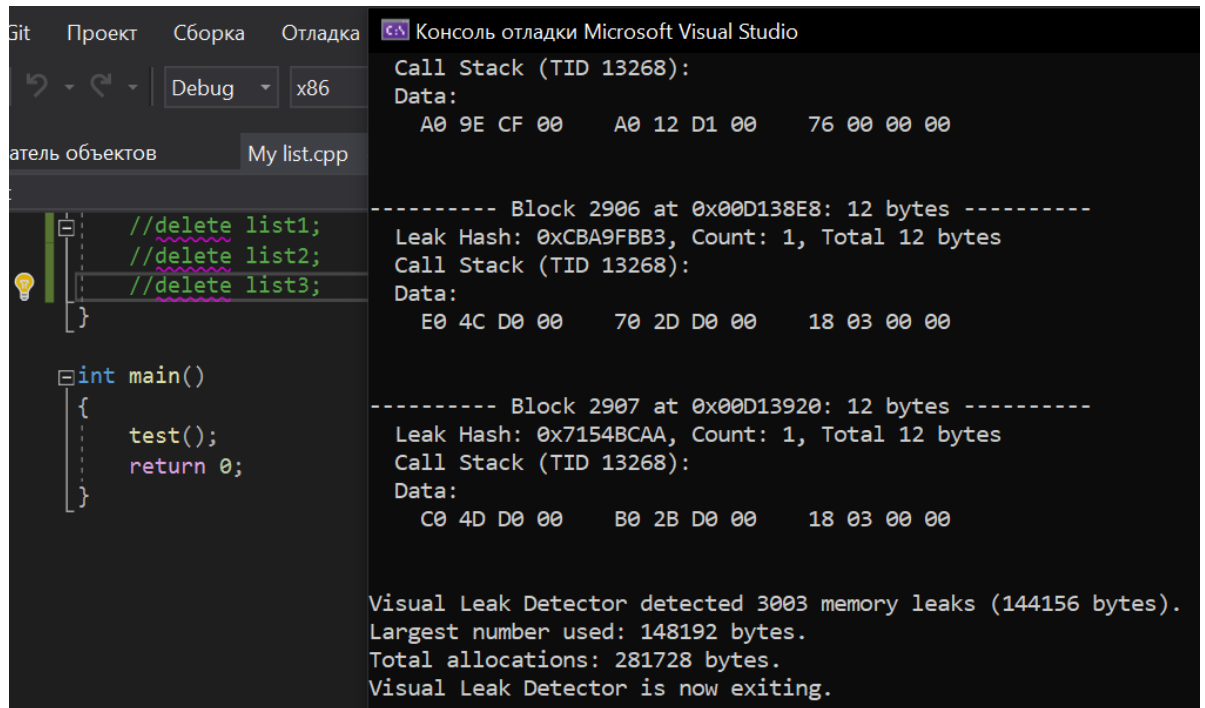
if (left < last) {
    temp_right = list->split_MemoryForNewListIsAllocatedThroughNew(left);
    list->merge(quickSort(temp_right));

    temp_right->reset();
    delete temp_right;
}
```

Рисунок 5 — С опытом придут более грамотные решения.



Отслеживать утечки памяти довольно сложно. Для того, чтобы проверять наличие утечек памяти, я скачал из интернета специальную библиотеку Visual Leak Detector. В режиме Debug сборки в консоль выводится информация об утерянных блоках памяти (рис. 6).



```
Git  Проект  Сборка  Отладка  Консоль отладки Microsoft Visual Studio
Debug  x86

My list.cpp

//delete list1;
//delete list2;
//delete list3;

int main()
{
    test();
    return 0;
}

Call Stack (TID 13268):
Data:
A0 9E CF 00  A0 12 D1 00  76 00 00 00

----- Block 2906 at 0x00D138E8: 12 bytes -----
Leak Hash: 0xCBA9FBB3, Count: 1, Total 12 bytes
Call Stack (TID 13268):
Data:
E0 4C D0 00  70 2D D0 00  18 03 00 00

----- Block 2907 at 0x00D13920: 12 bytes -----
Leak Hash: 0x71548CAA, Count: 1, Total 12 bytes
Call Stack (TID 13268):
Data:
C0 4D D0 00  B0 2B D0 00  18 03 00 00

Visual Leak Detector detected 3003 memory leaks (144156 bytes).
Largest number used: 148192 bytes.
Total allocations: 281728 bytes.
Visual Leak Detector is now exiting.
```

Рисунок 6 — Детектор обнаружил 3003 утечки памяти! В каждом списке — 1000 узлов. Плюс сами списки.

В процессе разработки программы все появляющиеся утечки памяти были добросовестно устранены.

Для отлова других потенциальных ошибок использовалась стандартная библиотека `cassert`. Если выражение в скобках возвращает `false`, программа принудительно завершается (рис. 7). Данный инструмент нужно применять только для отладки кода. В релизных программах нужно применять конструкции по обработке исключений, чтобы в случае ошибки в одном из модулей не «положить» всю систему целиком.

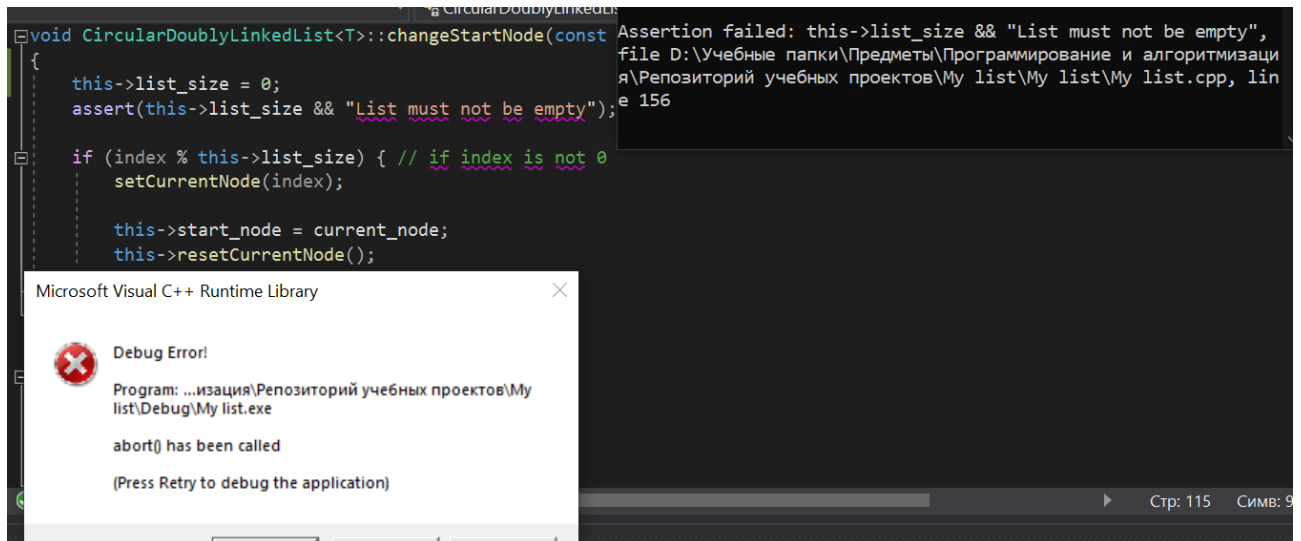


Рисунок 7 — Даже пустой *multi-character literal* всегда возвращает *true*. Поэтому можно добавить послание.

б) Помимо сортировок Шелла и Хоара, в коде была реализована сортировка слиянием. Перед тем, как начать анализ результатов, нужно переключиться в Release сборку. Производительность зависит не только от алгоритмов, но и от настроек конкретного компилятора. Например, в релизном режиме отключаются некоторые отладочные параметры. На временную сложность алгоритмов это никак не влияет.

Сначала сгенерируем три одинаковых списка со случайными значениями в диапазоне [100; 999] через современные методы рандомизации.

```
std::mt19937 rnd(time(0));
std::uniform_int_distribution<> uid(100, 999);

CircularDoublyLinkedList<int>* list1 = new CircularDoublyLinkedList<int>;
CircularDoublyLinkedList<int>* list2 = new CircularDoublyLinkedList<int>;
CircularDoublyLinkedList<int>* list3 = new CircularDoublyLinkedList<int>;
size_t size; std::cout << "List size: "; std::cin >> size;

for (size_t i = size; i != 0; --i) {
    int data = uid(rnd);
    list1->insertToLeft(data);
    list2->insertToLeft(data);
    list3->insertToLeft(data);
}
```

Затем посчитаем количество миллисекунд на выполнение каждого из алгоритмов через библиотеку chrono. Для удобства я написал простенький класс Timer.

```
class Timer
{
public:
    void start() {
        this->s = std::chrono::high_resolution_clock::now();
    }

    long long finish() {
        this->f = std::chrono::high_resolution_clock::now();
        return std::chrono::duration_cast<std::chrono::milliseconds>(this->f - this->s).count();
    }

private:
    std::chrono::time_point<std::chrono::steady_clock> s, f;
};
```

Расстановка таймеров в нужных местах.

```
Timer timer;
long long time1, time2, time3;
timer.start(); list1->mergeSort(); time1 = timer.finish();
timer.start(); list2->shellSort(); time2 = timer.finish();
timer.start(); list3 = quickSort(list3); time3 = timer.finish();
```

Проверим упорядоченность, «пробежав» по спискам после сортировки.

```
bool flag_fail1 = false, flag_fail2 = false, flag_fail3 = false;
std::cout << "\n\nChecking:";
for (size_t i = 1; i != size; ++i)
{
    flag_fail1 = (*list1)[i - 1] > (*list1)[i];
    flag_fail2 = (*list2)[i - 1] > (*list2)[i];
    flag_fail3 = (*list3)[i - 1] > (*list3)[i];
}
```

А ещё очень любопытно сравнить не только абсолютные значения, но и относительные.

```
std::cout << "Quick sort is " << (double)time1 / time3 << " times faster than merge
sort!\n";

std::cout << "Quick sort is " << (double)time2 / time3 << " times faster than shell
sort!\n";
std::cout << "Merge sort is " << (double)time2 / time1 << " times faster than shell
sort!\n\n";
```

Запущу тест на сто тысяч узлов в каждом списке (рис. 8).

```
999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999
999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999 999

Checking:
First list sorted (merge sort) in 51713 ms:      success!
Second list sorted (shell sort) in 983697 ms:   success!
Third list sorted (quick sort) in 46467 ms:     success!

Quick sort is 1.1129 times faster than merge sort!
Quick sort is 21.1698 times faster than shell sort!
Merge sort is 19.0222 times faster than shell sort!
```

Рисунок 8 — Если быстрая сортировка справилась за 46 секунд, то сортировке Шелла понадобилось 16 минут!

Сортировать можно не только числа. Вот пример со строками (рис. 9).

```
std::mt19937 rnd(time(0));
std::uniform_int_distribution<> uid(65, 90);

CircularDoublyLinkedList<std::string>* list1 = n;
CircularDoublyLinkedList<std::string>* list2 = n;
CircularDoublyLinkedList<std::string>* list3 = n;
size_t size; std::cout << "List size: "; std::cout << size << endl;

for (size_t i = size; i != 0; --i) {
    char a(uid(rnd)), b(uid(rnd)), c(uid(rnd));
    std::string data{ a, b, c };
    //list1->insertToLeft(data);
    //list2->insertToLeft(data);
    list3->insertToLeft(data);
}

Checking:
Third list sorted (quick sort) in 28 ms:      success!
D:\Учебные папки\Предметы\Программирование и алгоритмизация\Репозиторий учебных прое
оцесс 11924) завершил работу с кодом 0.
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Серв
томатически закрыть консоль при остановке отладки".
```

Рисунок 9 — Шаблонный класс и сортировки применимы для всех типов с перегруженными операторами сравнения.