МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА» (БГТУ им. В.Г. Шухова)

Кафедра программного обеспечения

вычислительной техники и автоматизированных

систем

Лабораторная работа №6

по дисциплине: ООП

тема: «Потоки в C++»

Выполнил: студент группы
ПВ-233
Мороз Роман Алексеевич

Проверили:
Морозов Данила Александрович

Белгород 2025

Лабораторная работа №7

Исключительные ситуации в C++

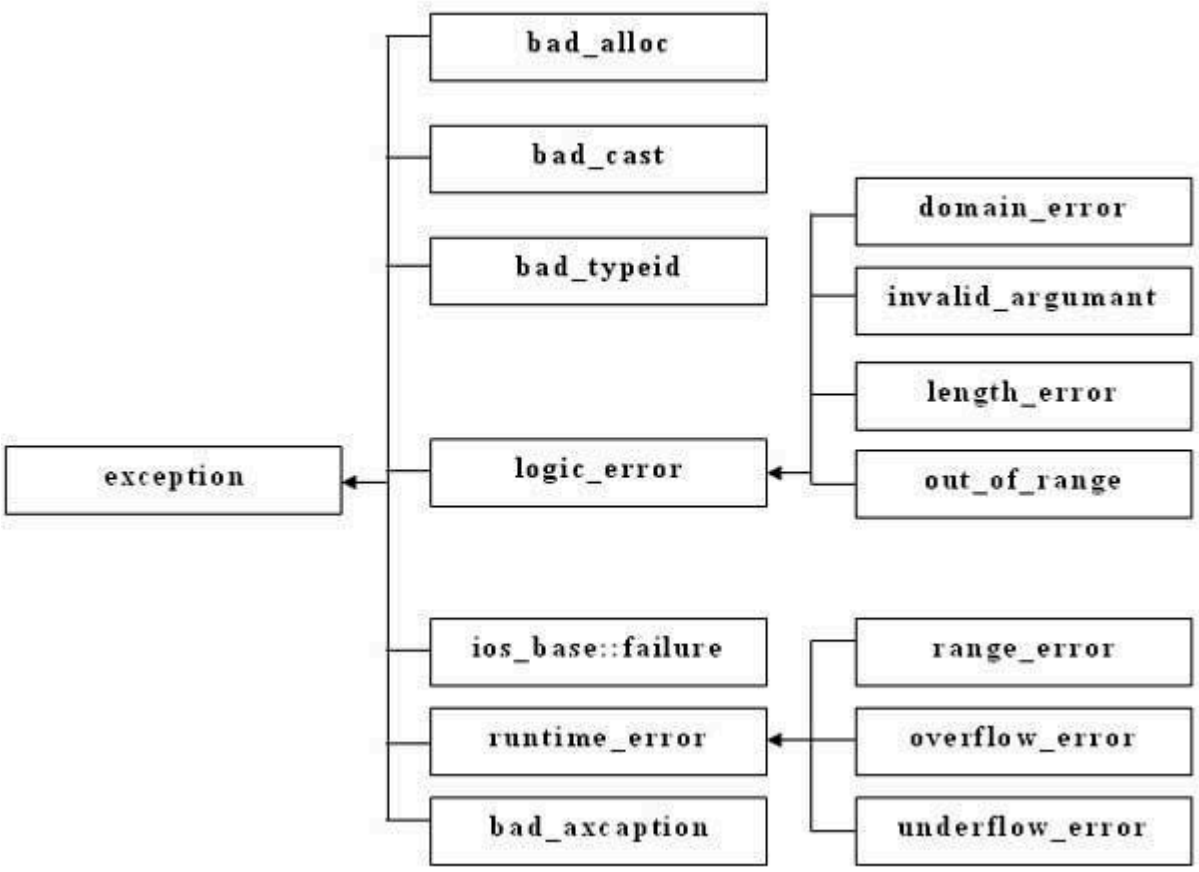
Цель работы: Получение теоретических знаний об исключительных ситуациях в C++. Получение практических навыков при работе с исключениями в C++.

Задания к лабораторной работе

1. Изучить теоретические сведения об исключениях в C++.
2. Изучить самостоятельно стандартные классы для исключений, предусмотренных в C++.
3. Разработать программу в соответствии с заданным вариантом задания.
4. Оформить отчет.

Краткие теоретические сведения

Стандартные классы исключений в C++:



Где

- bad_alloc, генерируемое операцией new;
- bad_cast, генерируемое операцией dynamic_cast;
- bad_typeid, генерируемое операцией typeid;
- bad_exception, генерируемое для обработки исключения, непредусмотренном в заголовке функции; исключения, порождаемые классами и алгоритмами STL;
- *исключение класса* invalid_argument сообщает о недопустимых значениях аргументов, например, когда битовые поля (массивы битов) инициализируются данными char, отличными от 0 и 1;
- *исключение класса* length_error сообщает о попытке выполнения операции, нарушающей ограничения на максимальный размер, например, при присоединении к строке слишком большого количества символов;
- *исключение класса* out_of_range сообщает о том, что аргумент не входит в интервал допустимых значений, например, при использовании неправильного индекса в коллекциях наподобие массивов или в строках;
- *исключение класса* domain_error сообщает об ошибке выхода за пределы области допустимых значений.;
- *исключение класса* range_error сообщает об ошибках выхода за пределы допустимого интервала во внутренних вычислениях;
- *исключение класса* overflow_error сообщает о математическом переполнении;
- *исключение класса* underflow_error сообщает о математической потере значимости.

Контрольные вопросы:

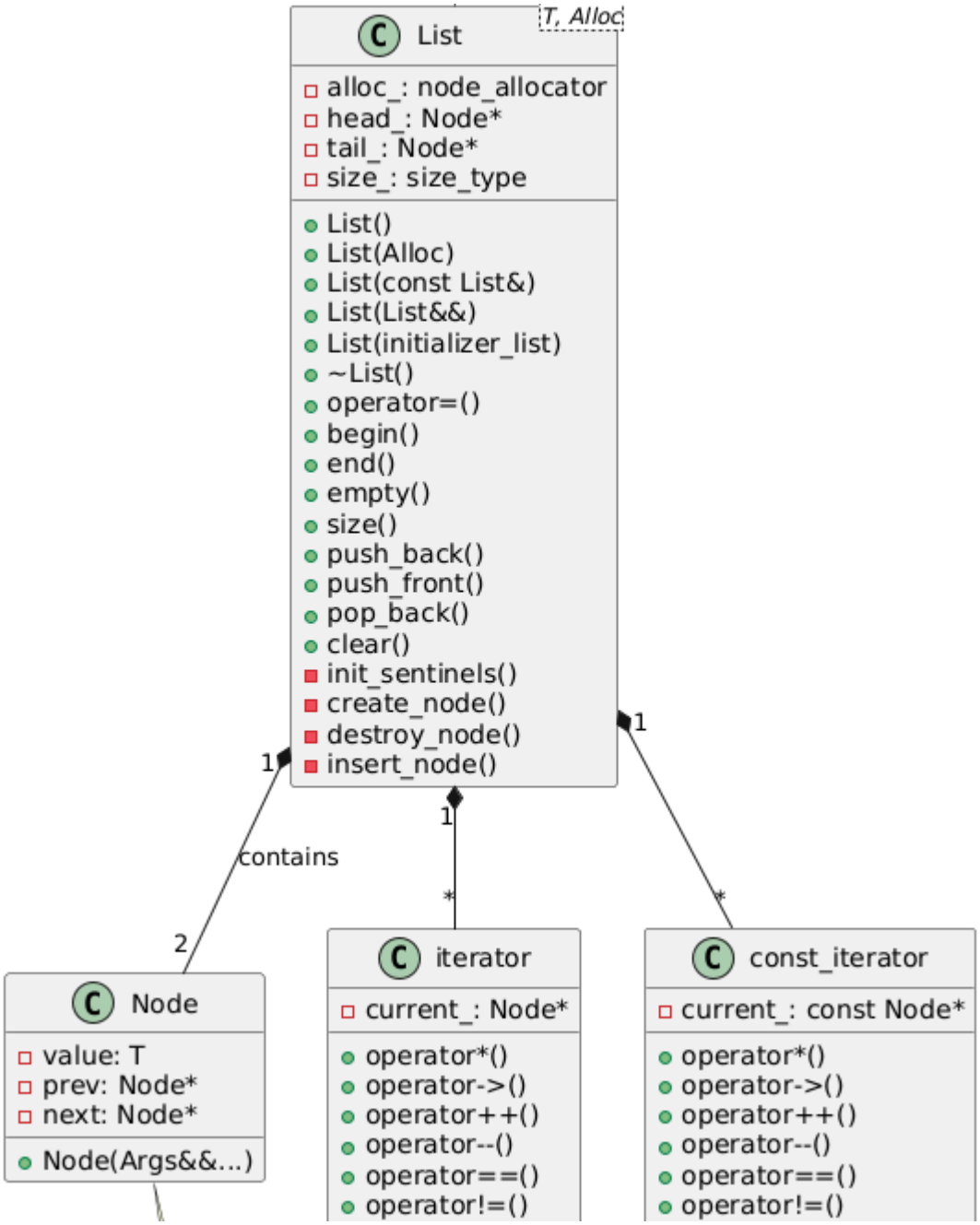
1. Виды ошибок при программировании.
2. Виды реализаций на возникающие ошибки.
3. Исключения это?
4. Описание обработки исключительных ситуаций C++.
5. Варианты параметров в блоке catch().

6. Обработка общих исключений.

Варианты задания

Вариант 9

Разработать класс “Список”. Предусмотреть операции создания, удаления, включения, исключения элементов, проверки на наличие элементов. Предусмотреть исключительные ситуации при работе со списками (возможность включения\исключения элемента) и другие.




```cpp
#ifndef SRC_LIST__HPP
#define SRC_LIST__HPP

#include <algorithm>
#include <initializer_list>
#include <memory>
#include <stdexcept>
#include <type_traits>

namespace mylist {
```

```cpp
template<typename T, typename Alloc = std::allocator<T>>
class List {
private:
    struct Node;

    using alloc_traits = std::allocator_traits<Alloc>;
    using node_allocator = typename alloc_traits::template
rebind_alloc<Node>;
    using node_alloc_traits = std::allocator_traits<node_allocator>;

public:
    using value_type = T;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using reference = value_type&;
    using const_reference = const value_type&;
    using pointer = typename alloc_traits::pointer;
    using const_pointer = typename alloc_traits::const_pointer;

    class iterator;
    class const_iterator;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator =
std::reverse_iterator<const_iterator>;

private:
    struct Node {
        value_type value;
        Node* prev;
        Node* next;

        template<typename... Args>
        Node(Node* p, Node* n, Args&&... args)
            : value(std::forward<Args>(args)...), prev(p), next(n) {}
    };

    node_allocator alloc_;
    Node* head_;
    Node* tail_;
    size_type size_;

public:
    class iterator {
```

```
        Node* current_;

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

        explicit iterator(Node* node = nullptr) : current_(node) {}

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

        inline iterator& operator++() {
            current_ = current_->next;
            return *this;
        }

        inline iterator operator++(int) {
            iterator tmp = *this;
            ++(*this);
            return tmp;
        }

        inline iterator& operator--() {
            current_ = current_->prev;
            return *this;
        }

        inline iterator operator--(int) {
            iterator tmp = *this;
            --(*this);
            return tmp;
        }

        inline bool operator==(const iterator& other) const { return
current_ == other.current_; }
        inline bool operator!=(const iterator& other) const { return
!(*this == other); }
    };

    class const_iterator {
```

```
        const Node* current_;

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

        explicit const_iterator(const Node* node = nullptr) :
current_(node) {}

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

        inline const_iterator& operator++() {
            current_ = current_->next;
            return *this;
        }

        inline const_iterator operator++(int) {
            const_iterator tmp = *this;
            ++(*this);
            return tmp;
        }

        inline const_iterator& operator--() {
            current_ = current_->prev;
            return *this;
        }

        inline const_iterator operator--(int) {
            const_iterator tmp = *this;
            --(*this);
            return tmp;
        }

        inline bool operator==(const const_iterator& other) const {
return current_ == other.current_; }
        inline bool operator!=(const const_iterator& other) const {
return !(*this == other); }
    };
```

```
    List() noexcept : alloc_(), head_(nullptr), tail_(nullptr), size_(0)
{
        init_sentinels();
    }

    explicit List(const Alloc& alloc) : alloc_(alloc), head_(nullptr),
tail_(nullptr), size_(0) {
        init_sentinels();
    }

    List(const List& other) :
alloc_(node_alloc_traits::select_on_container_copy_construction(other.a
lloc_)) {
        init_sentinels();
        for(const auto& item : other) {
            push_back(item);
        }
    }

    List(List&& other) noexcept : alloc_(std::move(other.alloc_)),
head_(other.head_),
                                  tail_(other.tail_), size_(other.size_) {
        other.head_ = other.tail_ = nullptr;
        other.size_ = 0;
    }

    List(std::initializer_list<value_type> init, const Alloc& alloc =
Alloc()) : alloc_(alloc) {
        init_sentinels();
        for(const auto& item : init) {
            push_back(item);
        }
    }

    ~List() {
        clear();
        destroy_node(head_);
        destroy_node(tail_);
    }

    List& operator=(const List& other) {
        if(this != &other) {
            clear();
```

```
            for(const auto& item : other) {
                push_back(item);
            }
        }
        return *this;
    }

    inline iterator begin() noexcept { return iterator(head_->next); }
    inline const_iterator begin() const noexcept { return
const_iterator(head_->next); }
    inline iterator end() noexcept { return iterator(tail_); }
    inline const_iterator end() const noexcept { return
const_iterator(tail_); }

    inline bool empty() const noexcept { return size_ == 0; }
    inline size_type size() const noexcept { return size_; }

    void push_back(const T& value) {
        insert_node(tail_->prev, tail_, value);
    }

    void push_front(const T& value) {
        insert_node(head_, head_->next, value);
    }

    void pop_back() noexcept {
        if(!empty()) {
            Node* to_remove = tail_->prev;
            to_remove->prev->next = tail_;
            tail_->prev = to_remove->prev;
            destroy_node(to_remove);
            --size_;
        }
    }

    void clear() noexcept {
        while(!empty()) {
            pop_back();
        }
    }

private:
    void init_sentinels() {
```

```
        head_ = create_node(nullptr, nullptr);
        tail_ = create_node(head_, nullptr);
        head_->next = tail_;
        tail_->prev = head_;
    }

    Node* create_node(Node* prev, Node* next) {
        Node* node = node_alloc_traits::allocate(alloc_, 1);
        try {
            node_alloc_traits::construct(alloc_, node, prev, next);
        } catch(...) {
            node_alloc_traits::deallocate(alloc_, node, 1);
            throw;
        }

        return node;
    }

    template<typename... Args>
    Node* create_node(Node* prev, Node* next, Args&&... args) {
        Node* node = node_alloc_traits::allocate(alloc_, 1);
        try {
            node_alloc_traits::construct(alloc_, node, prev, next,
std::forward<Args>(args)...);
        } catch(...) {
            node_alloc_traits::deallocate(alloc_, node, 1);
            throw;
        }

        return node;
    }

    void destroy_node(Node* node) noexcept {
        if(node) {
            node_alloc_traits::destroy(alloc_, node);
            node_alloc_traits::deallocate(alloc_, node, 1);
        }
    }

    void insert_node(Node* prev, Node* next, const T& value) {
        Node* new_node = create_node(prev, next, value);
        prev->next = new_node;
        next->prev = new_node;
```

```
        ++size_;
    }
};

} // namespace mylist

#endif // SRC_LIST__HPP
```

```
#include "list.hpp"
#include <iostream>


int main() {
    mylist::List<int> intList;

    intList.push_back(1);
    intList.push_back(2);
    intList.push_back(3);

    for(const auto& item : intList) {
        std::cout << item << ' ';
    }
    std::cout << std::endl;

    return 0;
}
```

Вывод: Получили теоретические знания об исключительных ситуациях в C++. Получили практические навыки при работе с исключениями в C++.