Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное учреждение высшего образования

«Пермский национальный исследовательский политехнический университет»

ПНИПУ

**Лабораторная работа по ООП**

**«№9»**

Выполнил:

студент группы РИС-23-1б

Кривошеин Александр Антонович

Проверила:

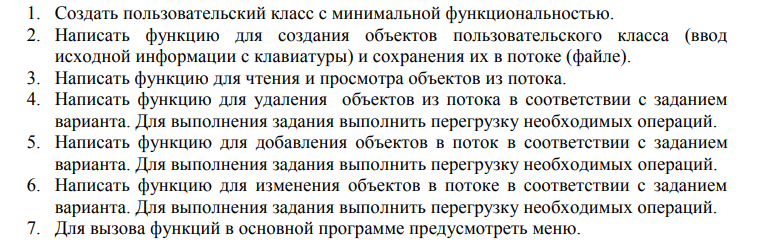
доцент кафедры ИТАС

Полякова Ольга Андреевна

2024 г.

**Разработка алгоритма**

**Постановка задачи:**



**Анализ задачи:**

Первая реализация обработки исключений:

1) Для реализации класса список необходимо создать структуру Node.

2) В классе список необходимо реализовать конструкторы, деструктор, функции вставки/ удаления первого/последнего элемента, а также операторы перегрузки.

3) Операции последовательного доступа можно реализовать посредством создания класса итератора. В классе должны быть конструкторы, деструктор, а также перегрузки операций.

[] – доступа по индексу;

int() – определение размера списка;

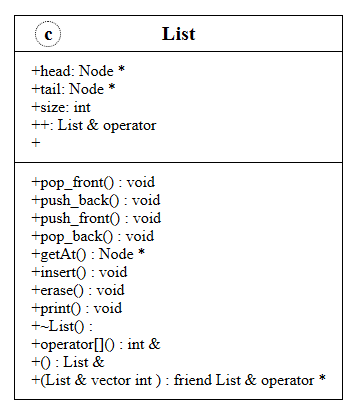
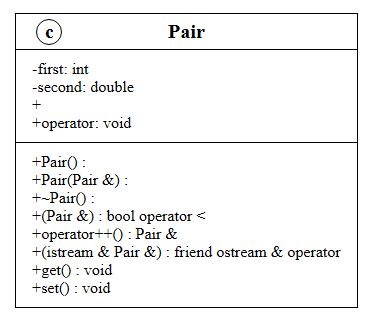
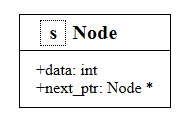
4) Исключительные ситуации генерируются:

В конструкторе с параметром при попытке создать вектор больше максимального размера;

В операции [] – при попытке обратиться к элементу с номером меньше 0 или больше текущего размера вектора;

6) Информация об исключительных ситуациях передается с помощью стандартного типа данных.

UML-диаграмма:



Код на С++:

#include <iostream>

#include <vector>

#include "List.h"

#include "List.cpp"

#include "iterator.cpp"

#include "iterator.h"

using namespace std;

int main()

{

    cout << "Enter size of list: ";

    try

    {

        int temp\_int;

        cin >> temp\_int;

        List list(temp\_int);

        for (Iterator iterator(list); iterator.getNode() != nullptr; iterator++) // вывод

        {

            cin >> temp\_int;

            iterator.getNode()->data = temp\_int;

        }

        list.print();

        cout << "First element of list: ";

        cout << list[0] << endl;

        cout << "Invalid operations:\n";

        cout << list[-3] << endl;

        // cout << list[60] << endl;

        // cout << list[list.size] << endl;

    }

    catch (int)

    {

        cout << "The size must be non-negative!";

    }

    catch (char)

    {

        cout << "Maximum size - 30!";

    }

    catch (float)

    {

        cout << "Index must be non-negative!";

    }

    catch (double)

    {

        cout << "Index must be in list-size range!";

    }

}

List.h:

#pragma once

#include <iostream>

#include <vector>

using namespace std;

struct Node

{

    int data = 0;

    Node \*next\_ptr = nullptr;

};

class List

{

public:

    Node \*head = nullptr;

    Node \*tail = nullptr;

    int size = 0;

    void pop\_front();

    void push\_back(int);

    void push\_front(int);

    void pop\_back();

    Node \*getAt(int);

    void insert(int, int);

    void erase(int);

    void print();

    ~List();

    List(int);

    int &operator[](int);

    List &operator()(int);

    List &operator+=(int);

    List &operator++(int);

    friend List &operator\*(List &a, vector<int> b);

};

List.cpp:

#pragma once

#include <iostream>

#include <vector>

#include "list.h"

#include "iterator.h"

using namespace std;

const int MAX\_SIZE = 30;

List::List(int k)

{

    size = k;

    for (int i = 0; i < k; i++)

    {

        this->push\_back(0);

    }

}

void List::pop\_front()

{

    if (head == nullptr)

        return;

    if (head == tail)

    {

        delete tail;

        head = tail = nullptr;

        return;

    }

    Node \*node = head;

    head = node->next\_ptr;

    delete node;

    --size;

}

void List::push\_back(int data)

{

    Node \*node = new Node;

    node->data = data;

    if (head == nullptr)

        head = node;

    if (tail != nullptr)

        tail->next\_ptr = node;

    tail = node;

    ++size;

}

void List::push\_front(int data)

{

    Node \*node = new Node;

    node->data = data;

    node->next\_ptr = head;

    head = node;

    if (tail == nullptr)

        tail = node;

    ++size;

}

void List::pop\_back()

{

    if (tail == nullptr)

        return;

    if (head == tail)

    {

        delete tail;

        head = tail = nullptr;

        return;

    }

    Node \*node = head;

    for (; node->next\_ptr != tail; node = node->next\_ptr)

        ;

    node->next\_ptr = nullptr;

    delete tail;

    tail = node;

    --size;

}

Node \*List::getAt(int k)

{

    if (k < 0)

        return 0;

    Node \*node = head;

    int n = 0;

    while (node && n != k && node->next\_ptr)

    {

        node = node->next\_ptr;

        n++;

    }

    if (n == k)

        return node;

    else

        return nullptr;

}

void List::insert(int k, int data)

{

    Node \*left = getAt(k);

    if (left == nullptr)

        return;

    Node \*right = left->next\_ptr;

    Node \*node = new Node;

    node->data = data;

    left->next\_ptr = node;

    node->next\_ptr = right;

    if (right == nullptr)

        tail = node;

    size++;

}

void List::erase(int k)

{

    if (k < 0)

        return;

    if (k == 0)

    {

        pop\_front();

        return;

    }

    Node \*left = getAt(k - 1);

    Node \*node = left->next\_ptr;

    if (node == nullptr)

        return;

    Node \*right = node->next\_ptr;

    left->next\_ptr = right;

    if (node == tail)

        tail = left;

    delete node;

    --size;

}

void List::print()

{

    for (Iterator iterator(\*this); iterator.getNode() != nullptr; iterator++) // вывод

        cout << iterator.getNode()->data << ' ';

    cout << endl

         << endl;

}

List &List::operator+=(int k)

{

    for (Iterator iterator(\*this); iterator.getNode() != nullptr; iterator++) // вывод

        iterator.getNode()->data += k;

    return \*this;

}

List &List::operator++(int)

{

    this->push\_back(0);

    return \*this;

}

List::~List()

{

    while (head != nullptr)

        pop\_front();

}

int &List::operator[](int a)

{

    if (a > this->size) throw double(1);

    if (a < 0) throw float(1.1);

    if (a > MAX\_SIZE) throw 'a';

    return getAt(a)->data;

}

List &List::operator()(int k)

{

    while (size > k)

    {

        this->pop\_back();

        size--;

    }

    while (size < k)

    {

        this->push\_back(0);

    }

    return \*this;

}

Iterator.cpp:

#pragma once

#include <iostream>

#include <string>

#include "iterator.h"

#include "list.h"

using namespace std;

Node \*Iterator::getNode()

{

    return this->iter;

}

Iterator::Iterator(List &a)

{

    this->iter = a.head;

}

Node \*Iterator::operator+(int a)

{

    Node \*iter = this->iter;

    for (int i = 0; i < a; ++i)

        iter = iter->next\_ptr;

    return iter;

}

Node \*Iterator::operator++()

{

    iter = iter->next\_ptr;

    return iter;

}

Node \*Iterator::operator++(int)

{

    iter = iter->next\_ptr;

    return iter;

}

Третья реализация обработки исключений:

Аналогично первой, но в throw необходимо передавать объект пользовательского класса Error, который является базовым для иерархии классов ошибок:

Error.cpp:

#pragma once

#include <iostream>

using namespace std;

class Error

{

public:

    virtual void what() {};

};

class IndexError : public Error

{

protected:

    string msg\_1;

public:

    IndexError()

    {

        msg\_1 = "Invalid index of list.";

    }

    virtual void what()

    {

        cout << msg\_1 << "\n";

    }

};

class ErrorIndexIsTooSmall : public IndexError

{

protected:

    string msg\_2;

public:

    ErrorIndexIsTooSmall()

    {

        IndexError();

        msg\_2 = "Index must be non-negative.";

    }

    virtual void what()

    {

        cout << msg\_1 << " " << msg\_2 << "\n\n";

    }

};

class ErrorIndexIsTooBig : public IndexError

{

protected:

    string msg\_2;

public:

    ErrorIndexIsTooBig()

    {

        IndexError();

        msg\_2 = "Index more than current list size.";

    }

    virtual void what() {

        cout << msg\_1 << " " << msg\_2 << "\n\n";

    }

};

class SizeError : public Error

{

protected:

    string msg\_1;

public:

    SizeError()

    {

        msg\_1 = "Invalid list size!";

    }

    virtual void what()

    {

        cout << msg\_1 << endl;

    }

};

class ErrorSizeIsTooSmall : public SizeError

{

protected:

    string msg\_2;

public:

    ErrorSizeIsTooSmall()

    {

        IndexError();

        msg\_2 = "Size must be non-negative!";

    }

    virtual void what()

    {

        cout << msg\_1 << " " << msg\_2 << "\n\n";

    }

};

class ErrorSizeIsTooBig : public SizeError

{

protected:

    string msg\_2;

public:

    ErrorSizeIsTooBig()

    {

        IndexError();

        msg\_2 = "The size is exceeded.";

    }

    virtual void what()

    {

        cout << msg\_1 << " " << msg\_2 << "\n\n";

    }

};

List.cpp:

#pragma once

#include <iostream>

#include <vector>

#include "list.h"

#include "iterator.h"

#include "errors.h"

using namespace std;

const int MAX\_SIZE = 30;

List::List(int k)

{

    if (k > MAX\_SIZE) throw ErrorSizeIsTooBig();

    if (k < 0) throw ErrorSizeIsTooSmall();

    size = k;

    for (int i = 0; i < k; i++)

    {

        this->push\_back(0);

    }

}

void List::pop\_front()

{

    if (head == nullptr)

        return;

    if (head == tail)

    {

        delete tail;

        head = tail = nullptr;

        return;

    }

    Node \*node = head;

    head = node->next\_ptr;

    delete node;

    --size;

}

void List::push\_back(int data)

{

    Node \*node = new Node;

    node->data = data;

    if (head == nullptr)

        head = node;

    if (tail != nullptr)

        tail->next\_ptr = node;

    tail = node;

    ++size;

}

void List::push\_front(int data)

{

    Node \*node = new Node;

    node->data = data;

    node->next\_ptr = head;

    head = node;

    if (tail == nullptr)

        tail = node;

    ++size;

}

void List::pop\_back()

{

    if (tail == nullptr)

        return;

    if (head == tail)

    {

        delete tail;

        head = tail = nullptr;

        return;

    }

    Node \*node = head;

    for (; node->next\_ptr != tail; node = node->next\_ptr)

        ;

    node->next\_ptr = nullptr;

    delete tail;

    tail = node;

    --size;

}

Node \*List::getAt(int k)

{

    if (k < 0)

        return 0;

    Node \*node = head;

    int n = 0;

    while (node && n != k && node->next\_ptr)

    {

        node = node->next\_ptr;

        n++;

    }

    if (n == k)

        return node;

    else

        return nullptr;

}

void List::insert(int k, int data)

{

    Node \*left = getAt(k);

    if (left == nullptr)

        return;

    Node \*right = left->next\_ptr;

    Node \*node = new Node;

    node->data = data;

    left->next\_ptr = node;

    node->next\_ptr = right;

    if (right == nullptr)

        tail = node;

    size++;

}

void List::erase(int k)

{

    if (k < 0)

        return;

    if (k == 0)

    {

        pop\_front();

        return;

    }

    Node \*left = getAt(k - 1);

    Node \*node = left->next\_ptr;

    if (node == nullptr)

        return;

    Node \*right = node->next\_ptr;

    left->next\_ptr = right;

    if (node == tail)

        tail = left;

    delete node;

    --size;

}

void List::print()

{

    for (Iterator iterator(\*this); iterator.getNode() != nullptr; iterator++) // вывод

        cout << iterator.getNode()->data << ' ';

    cout << endl

         << endl;

}

List &List::operator+=(int k)

{

    for (Iterator iterator(\*this); iterator.getNode() != nullptr; iterator++) // вывод

        iterator.getNode()->data += k;

    return \*this;

}

List &List::operator++(int)

{

    this->push\_back(0);

    return \*this;

}

List::~List()

{

    while (head != nullptr)

        pop\_front();

}

int &List::operator[](int a)

{

    if (a >= this->size) throw ErrorIndexIsTooBig();

    if (a < 0) throw ErrorIndexIsTooSmall();

    return getAt(a)->data;

}

List &List::operator()(int k)

{

    if (k > MAX\_SIZE) throw ErrorSizeIsTooBig();

    if (k < 0) throw ErrorSizeIsTooSmall();

    while (size > k)

    {

        this->pop\_back();

        size--;

    }

    while (size < k)

    {

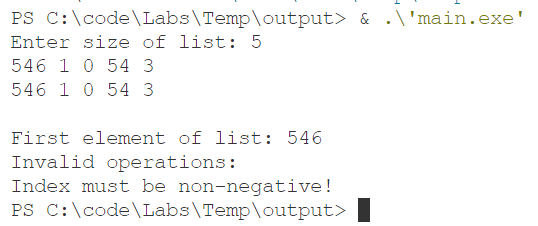
        this->push\_back(0);

    }

    return \*this;

}

Пример работы программы:



Контрольные вопросы

1. Что представляет собой исключение в С++?

Исключение – это непредвиденное или аварийное событие.

2. На какие части исключения позволяют разделить вычислительный процесс? Достоинства такого подхода?

Исключения позволяют разделить вычислительный процесс на 2 части:

1) обнаружение аварийной ситуации (неизвестно как обрабатывать);

2) обработка аварийной ситуации (неизвестно, где она возникла)

3. Какой оператор используется для генерации исключительной ситуации?

Исключение генерируется оператором throw <выражение>, где <выражение> -

• либо константа,

• либо переменная некоторого типа,

• либо выражение некоторого типа.

Тип объекта-исключения может быть как встроенным, так и определяемым пользователем

4. Что представляет собой контролируемый блок? Для чего он нужен?

Контролируемый блок — это блок кода, в котором создаются локальные переменные, инициализирующие объекты, и при выходе из блока автоматически вызываются деструкторы объектов.

Контролируемый блок используется для гарантированного выполнения действий при выходе из блока, независимо от того, какой путь был выбран при выходе.

try {//недопустимый размер множества, отрицательный размер List list\_3(-1);

cout << "list\_3" << endl;

cin >> list\_3;

cout << list\_3 << endl << endl;

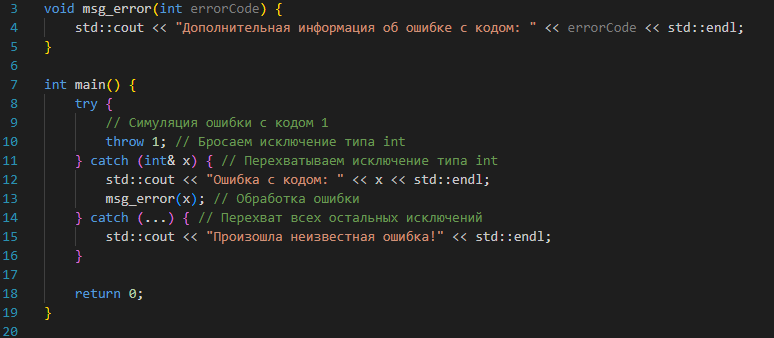
}

5. Что представляет собой секция-ловушка? Для чего она нужна?

Секция-ловушка (catch-блок) - это блок кода, который используется для

перехвата и обработки исключений, которые могут возникнуть в блоке try-

блоке.



6. Какие формы может иметь спецификация исключения в секции ловушке? В каких ситуациях используются эти формы?

- catch без аргумента типа: catch (...). Эта форма используется для перехвата любого исключения, которое не было перехвачено ранее. Она может быть по-лезна, например, для записи сообщения в журнал перед завершением про-граммы.

- catch с аргументом типа std::exception: catch (std::exception& e). Эта форма используется для перехвата исключений, производных от класса

std::exception. Она позволяет получить доступ к информации об исключении, например, к его сообщению, которое можно использовать для вывода пользователю.

- catch с аргументом типа T, где T - это класс исключения: catch (T& e).

Эта форма используется для перехвата конкретного класса исключения T. Она позволяет получить доступ к информации об исключении, которую можно ис-пользовать для его обработки.

- catch с несколькими аргументами типа: catch (T1& e1, T2& e2, ...). Эта форма используется для перехвата нескольких исключений разных типов. Она позво-ляет обработать каждое исключение по-разному в зависимости от его типа.

7. Какой стандартный класс можно использовать для создания собственной иерархии исключений?

class Error

8. Каким образом можно создать собственную иерархию исключений?

Для создания собственной иерархии исключений необходимо определить классы исключений, которые будут наследоваться от стандартного класса Error или его производных классов.

9. Если спецификация исключений имеет вид: void f1()throw(int,double); то какие исключения может прождать функция f1()?

Функция fl() может порождать исключения типа int и double

10. Если спецификация исключений имеет вид: void f1()throw(); то какие исключения может прождать функция f1()?

Функция fl() не может порождать никаких исключений. Это означает, что если в процессе выполнения функции возникнет исключение, то оно не будет обработано внутри этой функции и будет передано на уровень выше в стеке вызовов.

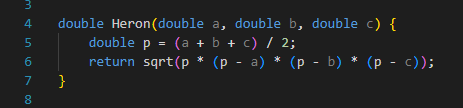
11. В какой части программы может генерироваться исключение?

В любой.

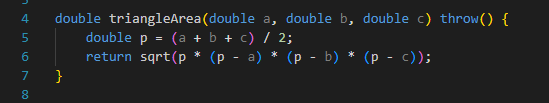
12. Написать функцию, которая вычисляет площадь треугольника по трем сторонам (формула Герона).

Функцию реализовать в 4 вариантах:

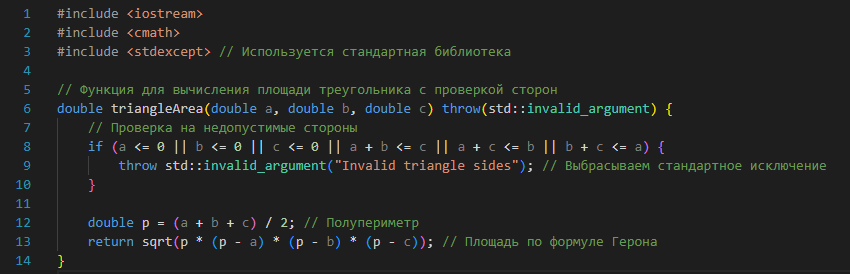
• без спецификации исключений;



• со спецификацией throw();



• с конкретной спецификацией с подходящим стандартным исключением;

• спецификация с собственным реализованным исключением.

