**Постановка задачи**

Решить поставленную задачу с использованием языка программирования С++, системы контроля версий Git и средств непрерывной интеграции, предоставляемых GitLab.

**S1. Стеки и очереди**

1. Ниже приведены интерфейсы шаблонов классов .Стек. и .Очередь

1 template< typename T >

2 class Queue {

3 public:

4 void push(T rhs);

5 T drop();

6 ...

7 };

8

9 template< typename T >

10 class Stack {

11 public:

12 void push(T rhs);

13 T drop();

14 ...

15 };

Переработайте интерфейс: исправьте и дополните его; сделайте безопасным. Реализуйте исправленные интерфейсы. Воспользуйтесь реализованными шаблонами классов для разбора арифметических выражений

1. Арифметические выражения содержатся в файле, имя которого задано параметром командной строки, по одному выражению на каждой строке.
   * Элементы выражения гарантировано разделены ровно одним пробелом (при этом выражения
   * могут быть неверными с точки зрения математики). Пример входного файла:

( 1 + 2 ) \* ( 3 – 4 )

1 + 3

( 10 / ( 2 + 3 ) % 4 )

4 \* 7 - 3

* + Пустые строки в файле пропускаются
  + Если параметр командной строки filename не задан, выражения читаются со стандартного ввода. Признаком конца ввода является EOF(на Linux: Ctrl + D | на Windows Ctrl + Z затем Enter)
  + Если арифметические выражения верны, результатом работы программы являются результаты вычисления выражений выведенные в обратном порядке (относительно порядка выражений) в одну строку, разделённые ровно одним пробелом
  + В арифмитических выражениях используются только бинарные операции: + — сложение, —вычитание, \* — умножение, / — деление, % — остаток от деления
  + В случае ошибки во время вычислений (например, операнд не является числом или арифметическое выражение неверно) программа должна завершаться с ненулевым кодом возврата и сообщением об ошибке

1. Работа должна быть выполнена в виде 1-го исполняемого файла, принимающего параметры следующим образом:

$ ./lab {filename}

filename представляет собой опциональной параметр. Поведение программы меняется в зависимости от того передан он или нет

**Приёмочные тесты**

Средствами непрерывной интеграции GitLab осуществляется автоматическая проверка результатов работы программы. В рамках настоящей работы предусмотрены следующие интеграционные тесты:

|  |  |  |
| --- | --- | --- |
| **#** | **Описание** | **Результат** |
| 1 | Сложение двух чисел с обработкой overflow | **Input:**  725 + 692  **Expected:**  1417 |
| **Input:**  9223372036854775806 + 1  **Expected:**  9223372036854775807 |
| **Input:**  9223372036854775807 + 1  **Expected:**  Сообщение «Overflow» и ненулевой код возврата |
| 2 | Деление положительных и отрицательных чисел с обработкой деления на ноль | **Input:**  585249 / 961  **Expected:**  609 |
| **Input:**  ( 0 - 510814 ) / 629  **Expected:**  -812 |
| **Input:**  5 / 0  **Expected:**  Сообщение «Division by 0» и ненулевой код возврата |
| 3 | Умножение двух чисел с обработкой overflow и underflow | **Input:**  934 \* 755  **Expected:**  705170 |
| **Input:**  ( 0 - 9223372036854775807 ) \* 2  **Expected:**  Сообщение «Underflow» и ненулевой код возврата |
| **Input:**  9223372036854775807 \* 2  **Expected:**  Сообщение «Overflow» и ненулевой код возврата |
| 4 | Вычитание положительных и отрицательных чисел с обработкой overflow и underflow | **Input:**  905 - 1184  **Expected:**  -279 |
| **Input:**  0 - 9223372036854775807 - 1  **Expected:**  -9223372036854775808 |
| **Input:**  0 - 9223372036854775808 – 1  **Expected:**  Сообщение «Underflow» и ненулевой код возврата |
| 5 | Взятие остатка от деления положительных и отрицательных чисел | **Input:**  364674 % 587  **Expected:**  147 |
| **Input:**  ( 0 - 240848 ) % 666  **Expected:**  244 |
| **Input:**  568364 % 941  **Expected:**  0 |
| 6 | Некорректное выражение | **Input:**  ( 1 - 2 ) + (  **Expected:**  Сообщение «Incorrect expression» и ненулевой код возврата |
| 7 | Одно число | **Input:**  413  **Expected:**  413 |
| **Input:**  ( 220 )  **Expected:**  220 |
| 8 | Комплексное выражение | **Input:**  ( 12 \* 12 ) - ( 12 + 13 )  **Expected:**  119 |
| 9 | Несколько комплексных арифметических выражений | **Input:**  1 + ( 2 - 1 )  ( 3 \* 2 ) / ( 3 - 1 )  **Expected:**  3 2 |
| **Input:**  1 + ( 2 - 1 )  ( 3 \* 2 ) / ( 3 - 1 )  **Expected:**  3 2 |
| **Input:**  ( 12 \* 12 ) - ( 12 + 13 )  **Expected:**  119 |

**Исходные тексты программы**

Файлы с исходными текстами лабораторной работы располагаются в корне общего проекта (полагаем <ROOT> для папки локального репозитория)

**./<ROOT>/nikiforova.ekaterina/S1/main.cpp**

#include <iostream>

#include <fstream>

#include <string>

#include <list.h>

#include "stack.h"

#include "queue.h"

#include "expressionPart.h"

#include "expression\_calculation.h"

int main(int argc, char\*\* argv)

{

std::ifstream fInput;

if (argc == 2)

{

fInput.open(argv[1]);

if (!fInput)

{

std::cerr << "File open error";

return 1;

}

}

std::istream& in = (argc == 2) ? fInput : std::cin;

nikiforova::Stack< long long > result;

std::string str;

while (std::getline(in, str))

{

if (str == "")

{

continue;

}

try

{

result.push(nikiforova::calculate(str));

}

catch (const std::exception& e)

{

std::cerr << e.what();

return 1;

}

}

if (!result.isEmpty())

{

std::cout << result.getTop();

result.drop();

while (!result.isEmpty())

{

std::cout << " " << result.getTop();

result.drop();

}

}

std::cout << "\n";

return 0;

}

**./<ROOT>/nikiforova.ekaterina/S1/expressionPart.h**

#ifndef EXPRESSIONPART\_H

#define EXPRESSIONPART\_H

namespace nikiforova {

enum class ExpressionType {

OpenedBracket,

ClosedBracket,

Operand,

Operation

};

class ExpressionPart {

public:

explicit ExpressionPart(char);

explicit ExpressionPart(long long);

~ExpressionPart() = default;

bool operator==(const ExpressionPart&) const;

bool operator!=(const ExpressionPart&) const;

ExpressionType getType() const;

long long getOperand() const;

char getOperation() const;

char getBracket() const;

private:

ExpressionType type\_;

union {

char mathSymbol\_;

long long operand\_;

};

};

}

#endif

**./<ROOT>/nikiforova.ekaterina/S1/expressionPart.cpp**

#include <stdexcept>

#include "expressionPart.h"

#include "expression\_calculation.h"

nikiforova::ExpressionPart::ExpressionPart(char s)

{

if (s == '(')

{

type\_ = ExpressionType::OpenedBracket;

mathSymbol\_ = s;

}

else if (s == ')')

{

type\_ = ExpressionType::ClosedBracket;

mathSymbol\_ = s;

}

else if (nikiforova::isMathOperation(s))

{

type\_ = ExpressionType::Operation;

mathSymbol\_ = s;

}

else

{

throw std::logic\_error("Not a bracket or math operation");

}

}

nikiforova::ExpressionPart::ExpressionPart(long long x):

type\_(ExpressionType::Operand),

operand\_(x)

{}

bool nikiforova::ExpressionPart::operator==(const ExpressionPart& rhs) const

{

return (type\_ == rhs.type\_) && (type\_ == ExpressionType::Operand ? operand\_ == rhs.operand\_ : mathSymbol\_ == rhs.mathSymbol\_);

}

bool nikiforova::ExpressionPart::operator!=(const ExpressionPart& rhs) const

{

return !(\*this == rhs);

}

nikiforova::ExpressionType nikiforova::ExpressionPart::getType() const

{

return type\_;

}

long long nikiforova::ExpressionPart::getOperand() const

{

if (type\_ != ExpressionType::Operand)

{

throw std::logic\_error("Not an operand");

}

return operand\_;

}

char nikiforova::ExpressionPart::getOperation() const

{

if (type\_ != ExpressionType::Operation)

{

throw std::logic\_error("Not an operation");

}

return mathSymbol\_;

}

char nikiforova::ExpressionPart::getBracket() const

{

if ((type\_ != ExpressionType::OpenedBracket) || (type\_ != ExpressionType::ClosedBracket))

{

throw std::logic\_error("Not a bracket");

}

return mathSymbol\_;

}

**./<ROOT>/nikiforova.ekaterina/S1/expression\_calculation.h**

#ifndef EXPRESSION\_CALCULATION\_H

#define EXPRESSION\_CALCULATION\_H

#include <string>

namespace nikiforova {

bool isMathOperation(const char);

long long calculate(std::string&);

}

#endif

**./<ROOT>/nikiforova.ekaterina/S1/expression\_calculation.cpp**

#include "expression\_calculation.h"

#include <string>

#include <limits>

#include "expressionPart.h"

#include "stack.h"

#include "queue.h"

#include "mathOperations.h"

namespace {

bool isSecondHasPriority(nikiforova::ExpressionPart ob1, nikiforova::ExpressionPart ob2)

{

using nikiforova::ExpressionPart;

bool a = (ob1 == ExpressionPart('+') || ob1 == ExpressionPart('-'));

bool b = ob2 == ExpressionPart('\*') || ob2 == ExpressionPart('/') || ob2 == ExpressionPart('%');

return a && b;

}

nikiforova::Queue< nikiforova::ExpressionPart > convertStringToInfix(std::string& str)

{

{

nikiforova::Queue< nikiforova::ExpressionPart > inputQueue;

for (unsigned int i = 0; i < str.length(); i++)

{

if (std::isspace(str[i]))

{

continue;

}

if (std::isalpha(str[i]))

{

throw std::logic\_error("Incorrect expression");

}

if (str[i] == '(' || str[i] == ')' || (nikiforova::isMathOperation(str[i]) && std::isspace(str[i + 1])))

{

inputQueue.push(nikiforova::ExpressionPart(str[i]));

}

else

{

std::string temp = "";

if (str[i] == '-')

{

temp.push\_back(str[i]);

i++;

}

while (isdigit(str[i]))

{

temp.push\_back(str[i]);

i++;

}

inputQueue.push(nikiforova::ExpressionPart(std::stoll(temp)));

}

}

return inputQueue;

}

}

nikiforova::Queue< nikiforova::ExpressionPart > convertInfixToPostfix(nikiforova::Queue< nikiforova::ExpressionPart > infixQueue)

{

nikiforova::Queue< nikiforova::ExpressionPart > postixQueue;

nikiforova::Stack< nikiforova::ExpressionPart > stack;

using nikiforova::ExpressionType;

using nikiforova::ExpressionPart;

long long countOpenBrackets = 0;

long long countClosedBrackets = 0;

long long countOperations = 0;

long long countOperands = 0;

while (!infixQueue.isEmpty())

{

if (infixQueue.getFront() == ExpressionPart('('))

{

stack.push(infixQueue.getFront());

infixQueue.drop();

countOpenBrackets++;

}

else if (infixQueue.getFront() == ExpressionPart(')'))

{

countClosedBrackets++;

if (countClosedBrackets > countOpenBrackets)

{

throw std::logic\_error("Incorrect expression");

}

while (stack.getTop() != ExpressionPart('('))

{

postixQueue.push(stack.getTop());

stack.getTop().getType() == ExpressionType::Operation ? countOperations++ : countOperands++;

stack.drop();

}

stack.drop();

infixQueue.drop();

}

else if (infixQueue.getFront().getType() == ExpressionType::Operand)

{

postixQueue.push(infixQueue.getFront());

infixQueue.drop();

countOperands++;

}

else

{

bool isInfixOperation = infixQueue.getFront().getType() == ExpressionType::Operation;

bool isStackOperation = !stack.isEmpty() ? stack.getTop().getType() == ExpressionType::Operation : 0;

while (!stack.isEmpty() && isInfixOperation && isStackOperation && (!isSecondHasPriority(infixQueue.getFront(), stack.getTop())))

{

postixQueue.push(stack.getTop());

countOperations++;

stack.drop();

stack.push(infixQueue.getFront());

infixQueue.drop();

isInfixOperation = infixQueue.getFront().getType() == ExpressionType::Operation;

isStackOperation = stack.getTop().getType() == ExpressionType::Operation;

}

stack.push(infixQueue.getFront());

infixQueue.drop();

}

}

while (!stack.isEmpty())

{

postixQueue.push(stack.getTop());

stack.getTop().getType() == ExpressionType::Operation ? countOperations++ : countOperands++;

stack.drop();

}

if ((countClosedBrackets != countOpenBrackets) || (countOperands - countOperations != 1))

{

throw std::logic\_error("Incorrect expression");

}

return postixQueue;

}

}

bool nikiforova::isMathOperation(char s)

{

return s == '+' || s == '-' || s == '\*' || s == '/' || s == '%';

}

long long nikiforova::calculate(std::string& str)

{

nikiforova::Queue< nikiforova::ExpressionPart > inputQueue;

inputQueue = ::convertStringToInfix(str);

nikiforova::Queue< nikiforova::ExpressionPart > postfixQueue;

postfixQueue = ::convertInfixToPostfix(inputQueue);

nikiforova::Stack< ExpressionPart > stack;

long long result = 0;

while (!postfixQueue.isEmpty())

{

if (postfixQueue.getFront().getType() == ExpressionType::Operand)

{

stack.push(postfixQueue.getFront());

postfixQueue.drop();

}

else if (postfixQueue.getFront().getType() == ExpressionType::Operation)

{

char operation = postfixQueue.getFront().getOperation();

postfixQueue.drop();

long long a = stack.getTop().getOperand();

stack.drop();

long long b = stack.getTop().getOperand();

stack.drop();

if (operation == '+')

{

result = nikiforova::sum(a, b);

}

else if (operation == '-')

{

result = nikiforova::sum(a \* (-1), b);

}

else if (operation == '\*')

{

result = nikiforova::mult(a, b);

}

else if (operation == '/')

{

if (a == 0)

{

throw std::logic\_error("Division by 0");

}

result = b / a;

}

else

{

result = nikiforova::mod(b, a);

}

stack.push(ExpressionPart(result));

}

}

if (!stack.isEmpty() && stack.getTop().getType() == ExpressionType::Operand)

{

result = stack.getTop().getOperand();

}

return result;

}

**./<ROOT>/nikiforova.ekaterina/S1/** **mathOperations.h**

#ifndef MATHOPERATIONS\_H

#define MATHOPERATIONS\_H

namespace nikiforova {

long long sum(long long a, long long b);

long long mult(long long a, long long b);

long long mod(long long a, long long b);

}

#endif

**./<ROOT>/nikiforova.ekaterina/S1/mathOperations.cpp**

#include "mathOperations.h"

#include <limits>

#include <stdexcept>

long long nikiforova::sum(long long a, long long b)

{

const long long max = std::numeric\_limits< long long >::max();

const long long min = std::numeric\_limits< long long >::min();

long long result = 0;

if (a > 0 && b > 0)

{

if (max - a < b)

{

throw std::overflow\_error("Overflow");

}

else

{

result = a + b;

}

}

else if (a < 0 && b < 0)

{

if (min - a > b)

{

throw std::underflow\_error("Underflow");

}

else

{

result = a + b;

}

}

else

{

result = a + b;

}

return result;

}

long long nikiforova::mult(long long a, long long b)

{

long long result = 0;

if ((a > 0 && b > 0) || (a < 0 && b < 0))

{

const long long max = std::numeric\_limits< long long >::max();

if (max / a < b)

{

throw std::overflow\_error("Overflow");

}

else

{

result = a \* b;

}

}

else

{

const long long min = std::numeric\_limits< long long >::min();

if (b < min / a)

{

throw std::underflow\_error("Underflow");

}

else

{

result = a \* b;

}

}

return result;

}

long long nikiforova::mod(long long a, long long b)

{

if (b == 0)

{

throw std::logic\_error("Division by 0");

}

if ((a > 0 && b > 0) || (a < 0 && b < 0))

{

return a % b;

}

else

{

return ((a % b + b) % b);

}

}

**./<ROOT>/nikiforova.ekaterina/S1/list.h**

#ifndef LIST\_H

#define LIST\_H

#include <iostream>

namespace nikiforova {

namespace detail {

template< typename T >

struct node\_t

{

T data\_;

node\_t\* next\_;

};

template< typename T >

class List {

public:

List();

List(const List&);

List(List&&) noexcept;

~List();

List& operator= (const List&);

List& operator= (List&&) noexcept;

size\_t size() const noexcept;

void pushFront(const T&);

void popFront();

void pushBack(const T&);

void swap(List&) noexcept;

void clear();

bool isEmpty() const noexcept;

const T& getFront() const;

const T& getBack() const;

protected:

node\_t< T >\* head\_;

node\_t< T >\* tail\_;

size\_t size\_;

};

template< typename T >

List< T >::List():

head\_(nullptr),

tail\_(nullptr),

size\_(0)

{}

template< typename T >

List< T >::List(const List< T >& x):

head\_(nullptr),

tail\_(nullptr),

size\_(0)

{

if (!x.isEmpty())

{

node\_t< T >\* srcPtr = x.head\_;

try

{

while (srcPtr)

{

pushBack(srcPtr->data\_);

srcPtr = srcPtr->next\_;

}

}

catch (...)

{

clear();

throw;

}

}

}

template< typename T >

List< T >::List(List< T >&& rhs) noexcept:

head\_(rhs.head\_),

tail\_(rhs.tail\_),

size\_(rhs.size\_)

{

rhs.tail\_ = nullptr;

rhs.head\_ = nullptr;

rhs.size\_ = 0;

}

template< typename T >

List< T >::~List()

{

clear();

}

template< typename T >

List< T >& List< T >::operator=(const List< T >& x)

{

if (this != std::addressof(x))

{

List< T > temp(x);

swap(temp);

}

return \*this;

}

template< typename T >

List< T >& List< T >::operator=(List< T >&& rhs) noexcept

{

if (this != std::addressof(rhs))

{

List< T > temp(std::move(rhs));

swap(temp);

}

return \*this;

}

template< typename T >

size\_t List< T >::size() const noexcept

{

return size\_;

}

template< typename T >

void List< T >::pushFront(const T& val)

{

head\_ = new node\_t< T >{ val, head\_ };

size\_++;

}

template< typename T >

void List< T >::popFront()

{

if (isEmpty())

{

throw std::logic\_error("Empty list");

}

node\_t< T >\* newHead = head\_->next\_;

if (head\_ == tail\_)

{

tail\_ = nullptr;

}

delete head\_;

head\_ = newHead;

size\_--;

}

template< typename T >

void List< T >::pushBack(const T& val)

{

if (isEmpty())

{

head\_ = new node\_t< T >{ val, nullptr };

tail\_ = head\_;

}

else

{

tail\_->next\_ = new node\_t< T >{ val, nullptr };

tail\_ = tail\_->next\_;

}

size\_++;

}

template< typename T >

void List< T >::swap(List< T >& x) noexcept

{

std::swap(head\_, x.head\_);

std::swap(tail\_, x.tail\_);

std::swap(size\_, x.size\_);

}

template< typename T >

void List< T >::clear()

{

while (!isEmpty())

{

popFront();

}

}

template< typename T >

bool List< T >::isEmpty() const noexcept

{

return !size\_;

}

template< typename T >

const T& List< T >::getFront() const

{

if (isEmpty())

{

throw std::logic\_error("Empty list");

}

return head\_->data\_;

}

template< typename T >

const T& List< T >::getBack() const

{

if (isEmpty())

{

throw std::logic\_error("Empty list");

}

return tail\_->data\_;

}

}

}

#endif

**./<ROOT>/nikiforova.ekaterina/S1/queue.h**

#ifndef QUEUE\_H

#define QUEUE\_H

#include "list.h"

namespace nikiforova {

template< typename T >

class Queue {

public:

void push(const T&);

void drop();

const T& getFront() const;

const T& getBack() const;

bool isEmpty() const noexcept;

size\_t getLenght() const;

private:

nikiforova::detail::List< T > list\_;

};

template< typename T >

void Queue< T >::push(const T& x)

{

list\_.pushBack(x);

}

template< typename T >

void Queue< T >::drop()

{

if (isEmpty())

{

throw std::logic\_error("Empty queue");

}

list\_.popFront();

}

template< typename T >

const T& Queue< T >::getFront() const

{

if (isEmpty())

{

throw std::logic\_error("Empty queue");

}

return list\_.getFront();

}

template< typename T >

const T& Queue< T >::getBack() const

{

if (isEmpty())

{

throw std::logic\_error("Empty queue");

}

return list\_.getBack();

}

template< typename T >

bool Queue< T >::isEmpty() const noexcept

{

return list\_.isEmpty();

}

template< typename T >

inline size\_t Queue< T >::getLenght() const

{

return list\_.size();

}

}

#endif

**./<ROOT>/nikiforova.ekaterina/S1/stack.h**

#ifndef STACK\_H

#define STACK\_H

#include "list.h"

namespace nikiforova {

template< typename T >

class Stack {

public:

const T& getTop() const;

void push(const T&);

void drop();

bool isEmpty() const noexcept;

size\_t getSize() const;

private:

nikiforova::detail::List< T > list\_;

};

template< typename T >

const T& Stack< T >::getTop() const

{

if (isEmpty())

{

throw std::logic\_error("Empty stack");

}

return list\_.getFront();

}

template< typename T >

void Stack< T >::push(const T& rhs)

{

list\_.pushFront(rhs);

}

template< typename T >

void Stack< T >::drop()

{

if (isEmpty())

{

throw std::logic\_error("Empty stack");

}

list\_.popFront();

}

template< typename T >

bool Stack< T >::isEmpty() const noexcept

{

return list\_.isEmpty();

}

template< typename T >

inline size\_t Stack< T >::getSize() const

{

return list\_.size();

}

}

#endif