МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования  
«Чувашский государственный университет им. И.Н. Ульянова»

Факультет информатики и вычислительной техники

Курсовая работа  
по дисциплине   
«Объектно-ориентированное программирование»  
на тему:  
«Лексический анализатор языка С++»

|  |
| --- |
| Выполнил: Трофимов Е.Э. студент группы: ИВТ-41-20  Проверил:  доц. к.т.н. Обломов И. А. Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Чебоксары, 2022

**Оглавление**

**Введение2**

**Постановка задачи3**

**Выбор метода, способа решения поставленной задачи и обоснование выбора** **4**

**Теоретический раздел** **6**

**Описание реализации лексического анализатора языка С++7**

**Заключение10**

**Список литературы11**

**Приложение 112**

**Приложение 213**

**Приложение 314**

**1.Введение**

Цель работы: Изучить лексический анализатор языка С++.

Лексический анализ – процесс аналитического разбора входной последовательности символов на распознанные группы – лексемы с целью получения на выходе идентифицированных последовательностей, называемых «токенами».

Лексемами языка С++ являются:

1. Идентификаторы
2. Константы
3. Ключевые слова
4. Знаки операций
5. Разделители

Лексический анализатор – часть компилятора, которая считывает исходный текст программы и выделяет в нём лексемы входного языка. Результатом работы лексического анализатора является перечень всех найденных в тексте исходной программы лексем.

**2. Постановка задачи**

Задача: Разработать лексический анализатор языка С++ .

Для выполнения задачи разобьём её на подзадачи:

1. Определить список разделителей.
2. Определить список ключевых слов.
3. Определить список знаков операций.
4. Реализовать функцию сканирования текста исходной программы и разбиения текста на лексемы.
5. Реализовать метод получения токена из лексемы
6. Реализовать функцию вывода результатов работы лексического анализатора

**3. Выбор метода, способа решения поставленной задачи и обоснование выбора**

Имеется 3 основных способа реализации лексического анализатора:

1. Использование генератора лексических анализаторов, работа которых основана на регулярных выражениях
2. Написание лексического анализатора на языке высокого уровня с использованием возможностей ввода/вывода данного языка для чтения и вывода входной информации
3. Написание лексического анализатора на языке ассемблера и явное управление процессом чтения входной информации

Преимущества первого способа:

1. Необходимость описания только лексем языка
2. Скорость получения конечных результатов

Недостатки первого способа:

1. Использование сложных регулярных выражений
2. Отсутствие гибкости

Преимущества написания лексического анализатора на ЯВУ:

1. Гибкость
2. Быстрота реализации
3. Повышение навыков реализации продукта
4. Отсутствие регулярных выражений

Недостатки написания лексического анализатора на ЯВУ:

1. Скорость получения конечных результатов ниже, чем при использовании генератора лексического анализатора

Преимущества написания ЛА на языке ассемблера:

1. Скорость работы программы

Недостатки:

1. Долгая реализация
2. Сложная реализация
3. Сложность поддержки продукта
4. Отсутствие гибкости

Мной было принято решение реализовать Лексический анализатор на языке С++ с использованием принципов объектно-ориентированного программирования.

**4. Теоретический раздел**

Объектно-ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования. ООП возникло в результате развития идеологии процедурного программирования, где данные и подпрограммы их обработки формально не связаны.

Принципы ООП:

1. Инкапсуляция
2. Абстракция
3. Наследование
4. Полиморфизм

Инкапсуляция – объединение данных и методов работы с данными в классе и скрытие реализации от пользователя.

Абстракция – выделение набора значимых характеристик объекта и исключение из рассмотрения незначимых.

Наследование – возможность описания нового класса на основе уже существующего с частично или полностью заимствованной функциональностью.

Полиморфизм – возможность объектов с одинаковой спецификацией иметь различную реализацию.

**5. Описание реализации лексического анализатора языка С++**

Алгоритм:

1. Посимвольное чтение текста исходной программы. (Приложение 1)
2. Разбиение текста на слова. (Приложение 1)
3. Получение токенов из лексем (Приложение 2)
4. Вывод токенов в выходной файл

Для удобства разработки файлы разбиваем по модулям.

Структура каталогов проекта:

* CourseWork
  + classes
    - TokensImmutableGroup.cpp
    - TokensImmutableGroup.h
    - TokensMutableGroup.cpp
    - TokensMutableGroup.h
  + consts
    - consts.cpp
    - consts.h
  + groups
    - groups.cpp
    - groups.h
  + tests
    - tests.cpp
    - tests.h
  + utils
    - utils.cpp
    - utils.h
  + main.cpp

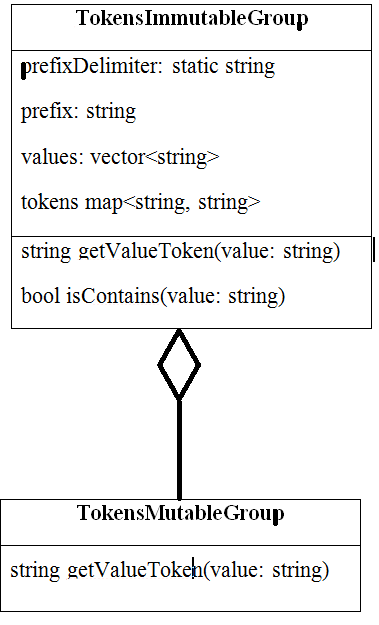
classes – классы, используемые в программе.

consts – константные значения.

groups – используется для инициализации групп токенов.

tests – тестовые функции для проверки правильности работы функций, методов на этапе разработки программы. Каждая функция в данном модуле имеет имя test<Название проверяемой функции или метода>. Использование тестов позволяет предотвратить ошибки, на начальных этапах, что упростит их исправление.

utils – полезные утилиты. В большей части – функции проверки принадлежности слова группе лексем.



*Рисунок 1. UML-диаграмма классов*

TokensImmutableGroup – группа токенов с постоянным составом (ключевые слова, разделители, знаки операций)

TokensMutableGroup – группа токенов с изменяемым составом (переменные, числовые константы, строковые константы, пользовательские типы данных). В данном классе переопределён метод getValueToken. Так, если значение уже в словаре, возвращает результат метода базового класса, иначе сначала заносим в словарь и только потом возвращаем токен.

Токен формируется следующим образом: прексГруппы + ”\_” + ИндексЛексемы

Префиксы групп:

* Ключевые слова – kw
* Разделители – sp
* Знаки операций – sg
* Переменные – vr
* Числовые константы – dc
* Строковые константы – sc
* Пользовательские типы данных – ut

**6. Заключение**

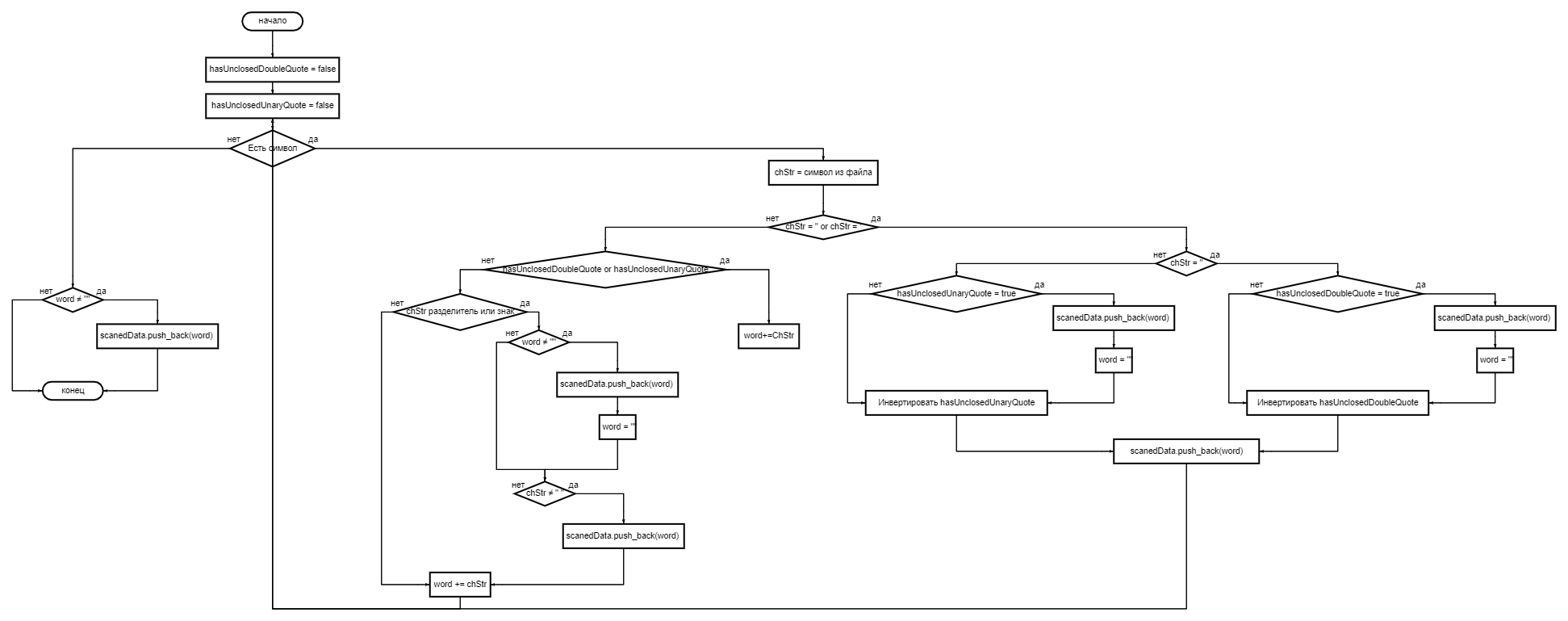
Лексический анализатор действительно имеет большое значение для компилятора, так как позволяет сократить объем информации, обрабатываемой на этапе синтаксического анализа. При этом некоторые задачи, требующие использования сложных вычислительных методов на этапе синтаксического анализа, могут быть решены более простыми методами на этапе лексического анализа. Стоит отметить, что использование концепции ООП значительно упрощает реализацию программы.

В ходе выполнения курсовой работы мною был изучен лексический анализатор языка С++, реализован собственный лексический анализатор языка С++ на языке высокого уровня С++ с использованием концепции объектно-ориентированного программирования.

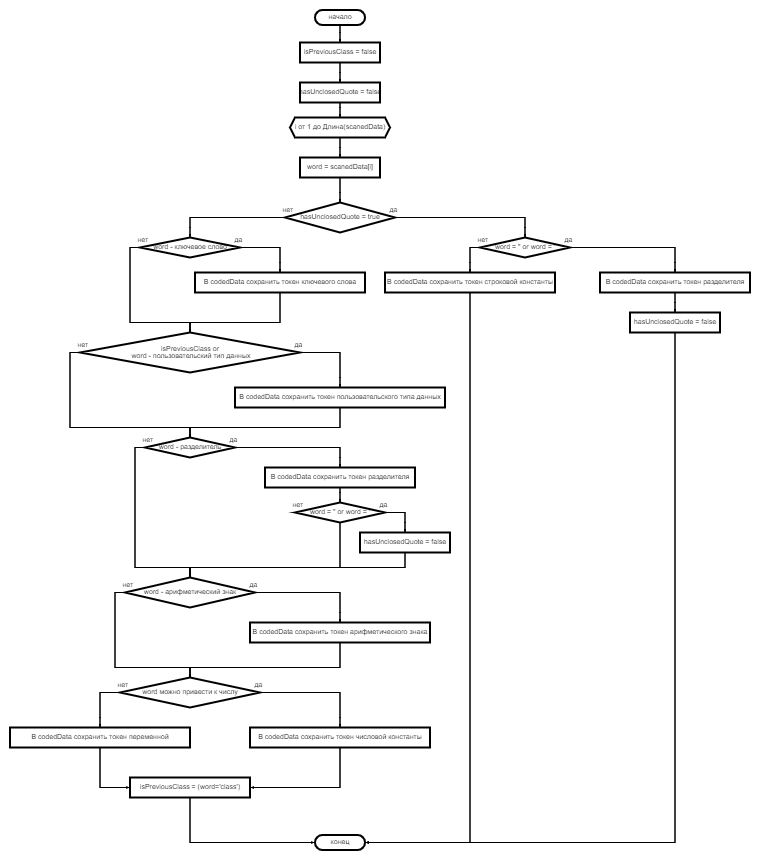
**7. Список литературы**

1. <https://ru.wikipedia.org/wiki/Лексический_анализ> (Дата обращения 01.05.2022)
2. <http://storage.piter.com/upload/contents/978549807153/978549807153_p.pdf> (Дата обращения 01.05.2022, 15.05.2022)
3. <https://ru.wikipedia.org/wiki/Объектно-ориентированное_программирование> (Дата обращения 15.05.2022)
4. <https://habr.com/ru/post/87205/> (Дата обращения 15.05.2022)

**Приложение 1**

**Приложение 2**

*Рисунок 2. Алгоритм посимвольного чтения файла и разбиения на слова*



*Рисунок 3. Алгоритм получения токенов из лексем*

**Приложение 3**

Текст программы:

*main.cpp*

#include <iostream>

#include <fstream>

#include "classes/TokensImmutableGroup.h"

#include "classes/TokensMutableGroup.h"

#include "consts/consts.h"

#include "groups/groups.h"

#include "utils/utils.h"

#include "tests/tests.h"

using namespace std;

// Отсанированные данные из входного файла. Храним глобально, чтобы легко добираться до данных

vector<string> scanedData;

// Закодированные данные

vector<string> codedData;

void scanFile() {

// Парсинг входного файла

// Входной файл

ifstream fileIn("test\_example.cpp");

char ch;

bool hasUnclosedDoubleQuote = false, hasUnclosedUnaryQuote = false;

string word;

while (fileIn.get(ch)) {

string chStr(1, ch);

if (isDoubleQuote(chStr) || isUnaryQuote(chStr)) {

// Сбрасываем слово только если есть незакрытая кавычка

if (isDoubleQuote(chStr)) {

if (hasUnclosedDoubleQuote) {

scanedData.push\_back(word);

word = "";

}

hasUnclosedDoubleQuote = !hasUnclosedDoubleQuote;

} else {

if (hasUnclosedUnaryQuote) {

scanedData.push\_back(word);

word = "";

}

hasUnclosedUnaryQuote = !hasUnclosedUnaryQuote;

}

scanedData.push\_back(chStr);

} else {

if (hasUnclosedDoubleQuote || hasUnclosedUnaryQuote) {

// Если есть незакрытые кавычки, на остальные символы внимания не обращаем

word += chStr;

} else {

if (isSpacer(chStr) || isSign(chStr) || isSpaceSymbol(chStr)) {

// Если видим разделитель, то сбрасываем текущее слово в контейнер спарсенных данных

if (!word.empty()) {

scanedData.push\_back(word);

word = "";

}

// Пробелы не рассматриваем

if (!isSpaceSymbol(chStr)) {

scanedData.push\_back(chStr);

}

} else {

word.push\_back(ch);

}

}

}

}

if (!word.empty()) {

// Закидываем последнее слово

scanedData.push\_back(word);

}

}

void codeData() {

// Кодирование данных

// Флаги предыдущих символов

bool isPreviousClass = false;

// Флаги кавычки

bool hasUnclosedQuote = false;

for (string word: scanedData) {

if (hasUnclosedQuote) {

if (isQuote(word)) {

codedData.push\_back(spacersGroup.getValueToken(word));

hasUnclosedQuote = false;

} else {

codedData.push\_back(stringConstsGroup.getValueToken(word));

}

} else {

if (isKeyWord(word)) {

// Ключевое слово языка С++

codedData.push\_back(keyWordsGroup.getValueToken(word));

} else if (isPreviousClass || isUserType(word)) {

// Пользовательский тип данных

codedData.push\_back(userTypesGroup.getValueToken(word));

} else if (isSpacer(word)) {

// Разделитель

codedData.push\_back(spacersGroup.getValueToken(word));

if (isQuote(word)) {

hasUnclosedQuote = true;

}

} else if (isSign(word)) {

// Арифметический знак

codedData.push\_back(arithmeticSignsGroup.getValueToken(word));

} else {

try {

// Если получится, значит числовая константа. Иначе - переменная

stoi(word);

codedData.push\_back(digitConstsGroup.getValueToken(word));

} catch (exception) {

// Не получилось. Переменная

codedData.push\_back(variablesGroup.getValueToken(word));

}

}

isPreviousClass = (word == "class");

}

}

}

void outCodedData() {

// Вывод закодированных данных

ofstream fileOut;

fileOut.open("result.txt");

for (string word: codedData) {

fileOut << word;

}

fileOut.close();

}

int main() {

system("chcp 65001");

prepare();

scanFile();

codeData();

outCodedData();

return 0;

}

TokensImmutableGroup.cpp

#include "TokensImmutableGroup.h"

TokensImmutableGroup::TokensImmutableGroup(string prefix\_, vector<string> values\_) {

prefix = move(prefix\_);

values = move(values\_);

this->fillTokens();

}

TokensImmutableGroup::TokensImmutableGroup(string prefix\_) {

prefix = move(prefix\_);

this->fillTokens();

}

void TokensImmutableGroup::fillTokens() {

// Заполнение словаря токенов

for (int i = 0; i < this->values.size(); i++) {

this->tokens[values[i]] = this->getToken(i);

}

}

string TokensImmutableGroup::getToken(int code) {

// Из значения в токен

return this->prefix + this->prefixDelimiter + to\_string(code);

}

string TokensImmutableGroup::getValueToken(const string &value) {

// Токен по значению

return this->tokens[value];

}

bool TokensImmutableGroup::isContains(const string &value) {

// Проверка вхождения значения в группу

return find(this->values.begin(), this->values.end(), value) != this->values.end();

}

string TokensImmutableGroup::prefixDelimiter = "\_";

*TokensImmutableGroup.h*

#include <iostream>

#include <string>

#include <utility>

#include <vector>

#include <map>

using namespace std;

#ifndef COURSEWORK\_TOKENSIMMUTABLEGROUP\_H

#define COURSEWORK\_TOKENSIMMUTABLEGROUP\_H

class TokensImmutableGroup {

// Класс для группы токенов, состав которой постоянен

protected:

void fillTokens();

string getToken(int code);

public:

// Разделитель между префиксом и кодом

static string prefixDelimiter;

// Префикс токена

string prefix;

// Значения, которые кодируем

vector<string> values;

// Словарь токенов

// token = prefix + prefixDelimiter + Код

map<string, string> tokens;

TokensImmutableGroup(string prefix\_, vector<string> values\_);

TokensImmutableGroup(string prefix\_);

string getValueToken(const string &value);

bool isContains(const string& value\_);

};

#endif //COURSEWORK\_TOKENSIMMUTABLEGROUP\_H

*TokensMutableGroup.h*

#include <iostream>

#include <string>

#include <utility>

#include <vector>

#include <map>

#include "TokensImmutableGroup.h"

#ifndef COURSEWORK\_TOKENSMUTABLEGROUP\_H

#define COURSEWORK\_TOKENSMUTABLEGROUP\_H

class TokensMutableGroup : public TokensImmutableGroup {

// Класс для группы токенов, состав которой непостоянен

public:

TokensMutableGroup(string prefix\_);

TokensMutableGroup(string prefix\_, vector<string> values\_);

string getValueToken(const string &value);

};

#endif //COURSEWORK\_TOKENSMUTABLEGROUP\_H

*TokensMutableGroup.cpp*

#include "TokensMutableGroup.h"

using namespace std;

TokensMutableGroup::TokensMutableGroup(string prefix\_) : TokensImmutableGroup(std::move(prefix\_)) {}

TokensMutableGroup::TokensMutableGroup(string prefix\_, vector<string> values\_) : TokensImmutableGroup(

std::move(prefix\_), std::move(values\_)) {}

string TokensMutableGroup::getValueToken(const string &value) {

// Токен по значению

if (this->isContains(value)) {

// Значение уже есть в группе, можем сразу вернуть токен

return TokensImmutableGroup::getValueToken(value);

} else {

// Сначала заносим в словарь токенов, потом возвращаем токен

this->values.push\_back(value);

this->tokens[value] = getToken(this->values.size() - 1);

return this->tokens[value];

}

}

*consts.h*

#include <string>

#include <vector>

using namespace std;

#ifndef COURSEWORK\_CONSTS\_H

#define COURSEWORK\_CONSTS\_H

// Список всех ключевых слов.

extern const vector<string> keyWords;

// Список всех разделителей

extern const vector<string> spacers;

// Список всех арифметических знаков

extern const vector<string> arithmeticSigns;

// Список знаков, которые могут использоваться в числовых константах

extern vector<string> digits;

// Префиксы групп

// Ключевые слова

extern string keyWordsGroupPrefix;

// Разделители

extern string spacersGroupPrefix;

// Арифметические знаки

extern string arithmeticSignsGroupPrefix;

// Переменные

extern string variablesGroupPrefix;

// Числовые константы

extern string digitConstsGroupPrefix;

// Строковые константы

extern string stringConstsGroupPrefix;

// Пользовательские типы данных

extern string userTypesGroupPrefix;

#endif //COURSEWORK\_CONSTS\_H

consts.cpp

#include <vector>

#include "consts.h"

using namespace std;

// Списки значений для постоянных групп

// Список всех ключевых слов.

const vector<string> keyWords = {

"alignas", "alignof", "and", "and\_eq", "asm", "auto", "bitand", "bitor", "bool", "break", "case", "catch",

"char", "char8\_t", "char16\_t", "char32\_t", "class", "compl", "concept", "const", "const\_cast", "consteval",

"constexpr", "constinit", "continue", "co\_await", "co\_return", "co\_yield", "decltype", "default", "delete",

"do", "double", "dynamic\_cast", "else", "enum", "explicit", "export", "extern", "false", "float", "for",

"friend", "goto", "if", "inline", "int", "long", "mutable", "namespace", "new", "noexcept", "not", "not\_eq",

"nullptr", "operator", "or", "or\_eq", "private", "protected", "public", "register", "reinterpret\_cast",

"requires", "return", "short", "signed", "sizeof", "static", "static\_assert", "static\_cast", "struct", "switch",

"template", "this", "thread\_local", "throw", "true", "try", "typedef", "typeid", "typename", "union",

"unsigned", "using", "virtual", "void", "volatile", "volatile", "while", "xor", "xor\_eq", "include",

};

// Список разделителей

const vector<string> spacers = {

"(", ")", "{", "}", "[", "]", "\"", "'", ":", ";", ",", "#"

};

// Список арифметических знаков

const vector<string> arithmeticSigns = {

"+", "-", "<", ">", "=", "!", "&", "|"

};

// Список знаков, которые могут использоваться в числовых константах

vector<string> digits;

// Префиксы групп. Правило: длина префикса равна 2

//Ключевые слова -> key words -> kw

string keyWordsGroupPrefix = "kw";

// Разделители -> spacers -> sp

string spacersGroupPrefix = "sp";

// Арифметические знаки -> arithmetic signs -> sg

string arithmeticSignsGroupPrefix = "sg";

// Переменные -> variables -> vr

string variablesGroupPrefix = "vr";

// Числовые константы -> digit consts -> dc

string digitConstsGroupPrefix = "dc";

// Строковые константы -> string consts -> sc

string stringConstsGroupPrefix = "sc";

// Пользовательские типы данных -> user data types -> ut

string userTypesGroupPrefix = "ut";

*groups.h*

#include "../classes/TokensImmutableGroup.h"

#include "../classes/TokensMutableGroup.h"

#ifndef COURSEWORK\_GROUPS\_H

#define COURSEWORK\_GROUPS\_H

// Неизменяемые группы

// Ключевые слова

extern TokensImmutableGroup keyWordsGroup;

// Разделители

extern TokensImmutableGroup spacersGroup;

// Арифметические знаки

extern TokensImmutableGroup arithmeticSignsGroup;

// Изменяемые группы

// Переменные

extern TokensMutableGroup variablesGroup;

// Числовые константы

extern TokensMutableGroup digitConstsGroup;

// Строковые константы

extern TokensMutableGroup stringConstsGroup;

// Пользовательские типы данных

extern TokensMutableGroup userTypesGroup;

#endif //COURSEWORK\_GROUPS\_H

*groups.cpp*

#include "groups.h"

#include "../consts/consts.h"

// Неизменяемые группы

// Ключевые слова

TokensImmutableGroup keyWordsGroup(keyWordsGroupPrefix, keyWords);

// Разделители

TokensImmutableGroup spacersGroup(spacersGroupPrefix, spacers);

// Арифметические знаки

TokensImmutableGroup arithmeticSignsGroup(arithmeticSignsGroupPrefix, arithmeticSigns);

// Изменяемые группы

// Переменные

TokensMutableGroup variablesGroup(variablesGroupPrefix);

// Числовые константы

TokensMutableGroup digitConstsGroup(digitConstsGroupPrefix);

// Строковые константы

TokensMutableGroup stringConstsGroup(stringConstsGroupPrefix);

// Пользовательские типы данных

TokensMutableGroup userTypesGroup(userTypesGroupPrefix);

tests.h

#ifndef COURSEWORK\_TESTS\_H

#define COURSEWORK\_TESTS\_H

// "Тестирование" кода

extern void test();

// Тестирование неизменяемой группы токенов + тестирование группы токенов ключевых слов

extern void testKeyWordsGroup();

// Разделители

extern void testSpacersGroup();

// Арифметические знаки

extern void testArithmeticSignsGroup();

// Переменные

extern void testVariablesGroup();

// Числовые константы

extern void testDigitConstsGroup();

// Строковые константы

extern void testStringConstsGroup();

// Пользовательские типы данных

extern void testUserTypesGroup();

// Тестирование заполнения знаков для числовых констант

extern void testDigitsPreparation();

// Тестирование функции, определяющей, является ли символ разделителем

extern void testIsSpacer();

// Тестирование функции, определяющей, является ли символ арифметическим знаком

extern void testIsSign();

// Тестирование функции, определяющей, является ли символ знаком, который может использоваться в числовых константах

extern void testIsDigit();

// Тестирование функции, определяющей, является ли символ одинарной кавычкой

extern void testIsUnaryQuote();

// Тестирование функции, определяющей, является ли символ двойной кавычкой

extern void testIsDoubleQuote();

// Тестирование функции, определяющей, является ли слово ключевым словом языка С++

extern void testIsKeyWord();

#endif //COURSEWORK\_TESTS\_H

*tests.cpp*

#include "../classes/TokensImmutableGroup.h"

#include "../classes/TokensMutableGroup.h"

#include "../consts/consts.h"

#include "../groups/groups.h"

#include "../utils/utils.h"

#include "tests.h"

void test() {

// Тестировать лучше всё сразу, чтоб быть уверенным, что ничего не сломалось

// Группы

testKeyWordsGroup();

testSpacersGroup();

testArithmeticSignsGroup();

testStringConstsGroup();

testVariablesGroup();

testDigitConstsGroup();

testUserTypesGroup();

// Прочие утилиты

testDigitsPreparation();

// Функции проверки вхождения в группу токенов или логическую группу

testIsSpacer();

testIsSign();

testIsDigit();

testIsUnaryQuote();

testIsDoubleQuote();

testIsKeyWord();

}

void testKeyWordsGroup() {

// Тестирование неизменяемой группы токенов + тестирование группы токенов ключевых слов

// Ожидается: kw\_0 kw\_1

// Проверено

cout << keyWordsGroup.getValueToken("alignas") << ' ';

cout << keyWordsGroup.getValueToken("alignof");

cout << endl;

}

void testSpacersGroup() {

// Тестирование группы токенов для разделителей

// Ожидается: sp\_1 sp\_7 sp\_0

// Проверено

cout << spacersGroup.getValueToken(")") << ' ';

cout << spacersGroup.getValueToken("\"") << ' ';

cout << spacersGroup.getValueToken("(") << ' ';

cout << endl;

}

void testArithmeticSignsGroup() {

// Тестирование группы токенов для арифметических знаков

// Ожидается: sg\_0 sg\_7

// Проверено

cout << arithmeticSignsGroup.getValueToken("+") << ' ';

cout << arithmeticSignsGroup.getValueToken("|") << ' ';

cout << endl;

}

void testStringConstsGroup() {

// Тестирование изменяемой группы токенов + тестирование группы токенов для строковых констант

// Ожидается: sc\_0 sc\_1 sc\_1

// Проверено

cout << stringConstsGroup.getValueToken("abc") << ' ';

cout << stringConstsGroup.getValueToken("erf") << ' ';

cout << stringConstsGroup.getValueToken("erf");

cout << endl;

}

void testVariablesGroup() {

// Тестирование группы токенов для переменных

// Ожидается: vr\_0 vr\_1 vr\_0

// Проверено

cout << variablesGroup.getValueToken("ab") << ' ';

cout << variablesGroup.getValueToken("cd") << ' ';

cout << variablesGroup.getValueToken("ab") << ' ';

cout << endl;

}

void testDigitConstsGroup() {

// Тестирование группы токенов для целочисленных констант

// Ожидается: dc\_0 dc\_1 dc\_2 dc\_1

// Проверено

cout << digitConstsGroup.getValueToken(to\_string(50)) << ' ';

cout << digitConstsGroup.getValueToken(to\_string(70)) << ' ';

cout << digitConstsGroup.getValueToken(to\_string(80)) << ' ';

cout << digitConstsGroup.getValueToken(to\_string(70)) << ' ';

cout << endl;

}

void testUserTypesGroup() {

// Тестирование группы токенов для пользовательских типов данных

// Ожидается: ut\_0 ut\_1

// Проверено

cout << userTypesGroup.getValueToken("TokensMutableGroup") << ' ';

cout << userTypesGroup.getValueToken("TokensImmutableGroup") << ' ';

cout << endl;

}

void testDigitsPreparation() {

// Тестирование заполнения знаков для числовых констант

// Ожидается: 0 5 9 .

// Проверено

cout << digits[0] << ' ' << digits[5] << ' ' << digits[9] << ' ' << digits[10];

cout << endl;

}

void testIsSpacer() {

// Тестирование функции, определяющей, является ли символ разделителем

// Ожидается: 1 0 0

// Проверено

cout << isSpacer("(") << ' ';

cout << isSpacer("f") << ' ';

cout << isSpacer(" ") << ' ';

cout << endl;

}

void testIsSign() {

// Тестирование функции, определяющей, является ли символ арифметическим знаком

// Ожидается: 1 0 1

// Проверено

cout << isSign("+") << ' ';

cout << isSign("f") << ' ';

cout << isSign("-") << ' ';

cout << endl;

}

void testIsDigit() {

// Тестирование функции, определяющей, является ли символ знаком, который может использоваться в числовых константах

// Ожидается: 1 1 0]

// Проверено

cout << isDigit("5") << ' ';

cout << isDigit(".") << ' ';

cout << isDigit("f") << ' ';

cout << endl;

}

void testIsUnaryQuote() {

// Тестирование функции, определяющей, является ли символ одинарной кавычкой

// Ожидается: 1 0

// Проверено

cout << isUnaryQuote("\'") << ' ';

cout << isUnaryQuote("\"") << ' ';

cout << endl;

}

void testIsDoubleQuote() {

// Тестирование функции, определяющей, является ли символ одинарной кавычкой

// Ожидается: 0 1

// Проверено

cout << isDoubleQuote("\'") << ' ';

cout << isDoubleQuote("\"") << ' ';

cout << endl;

}

void testIsKeyWord() {

// Тестирование функции, определяющей, является ли слово ключевым словом языка С++

// Ожидается: 1 0 1

cout << isKeyWord("class") << ' ';

cout << isKeyWord("abc") << ' ';

cout << isKeyWord("throw") << ' ';

cout << endl;

}

*utils.h*

#include <string>

#include "../classes/TokensImmutableGroup.h"

using namespace std;

#ifndef COURSEWORK\_UTILS\_H

#define COURSEWORK\_UTILS\_H

// Подготовка к выполнению программы

extern void prepare();

// Заполнение списка знаков, которые могут использоваться в числовых константах

extern void fillDigits();

// Базовая функция для проверки вхождения символа в определенную группу

extern bool baseInGroupChecker(TokensImmutableGroup, string);

// Является ли символ разделителем

extern bool isSpacer(string);

// Является ли символ арифметическим знаком

extern bool isSign(string);

// Является ли слово ключевым словом языка С++

extern bool isKeyWord(string);

// Является ли слово пользовательским типом данных

extern bool isUserType(string);

// Является ли символ знаком, который может использоваться в числовых константах

extern bool isDigit(string);

// Является ли символ одинарной кавычкой

extern bool isUnaryQuote(string);

// Является ли символ двойной кавычкой

extern bool isDoubleQuote(string);

// Является ли символ кавычкой

extern bool isQuote(string);

// Является ли символ пробельным

extern bool isSpaceSymbol(string);

#endif //COURSEWORK\_UTILS\_H

*utils.cpp*

#include "../classes/TokensImmutableGroup.h"

#include "utils.h"

#include "../consts/consts.h"

#include "../groups/groups.h"

// Подготовка к выполнению программы

void prepare() {

fillDigits();

}

// Заполнение списка знаков, которые могут использоваться в числовых константах

void fillDigits() {

for (int i = 0; i < 10; i++) {

digits.push\_back(to\_string(i));

}

digits.push\_back(".");

}

// Функции, необходимые при парсинге входного файла

// Базовая функция для проверки вхождения символа в определенную группу

bool baseInGroupChecker(TokensImmutableGroup group, string value) {

return group.isContains(value);

}

// Является ли символ разделителем

bool isSpacer(string value) {

return baseInGroupChecker(spacersGroup, value);

}

// Является ли символ арифметическим знаком

bool isSign(string value) {

return baseInGroupChecker(arithmeticSignsGroup, value);

}

// Является ли слово ключевым словом языка С++

bool isKeyWord(string value) {

return baseInGroupChecker(keyWordsGroup, value);

}

// Является ли слово пользовательским типом данных

bool isUserType(string value) {

return baseInGroupChecker(userTypesGroup, value);

}

// Является ли символ знаком, который может использоваться в числовых константах

bool isDigit(string value) {

return find(digits.begin(), digits.end(), value) != digits.end();

}

// Является ли символ одинарной кавычкой

bool isUnaryQuote(string value) {

return value == "\'";

}

// Является ли символ двойной кавычкой

bool isDoubleQuote(string value) {

return value == "\"";

}

// Является ли символ кавычкой

bool isQuote(string value) {

return isUnaryQuote(value) || isDoubleQuote(value);

}

// Является ли символ пробельным

bool isSpaceSymbol(string value) {

return (value == " ") || (value == "\n") || (value == "\t");

}

Пример выполнения программы

Исходный текст программы:

#include <iostream>

using namespace std;

class A {

};

int main() {

int a\_1;

string str;

str = "abc.\_123 gjghj";

s = 'f';

a\_1 = 5;

float b = 1.45;

for (int i = 1; i < 10; i++) {

if (i != 7) {

cout << i;

}

}

return 0;

}

Результат выполнения программы:

sp\_11kw\_92sg\_2vr\_0sg\_3kw\_84kw\_49vr\_1sp\_9kw\_16ut\_0sp\_2sp\_3sp\_9kw\_46vr\_2sp\_0sp\_1sp\_2kw\_46vr\_3sp\_9vr\_4vr\_5sp\_9vr\_5sg\_4sp\_6sc\_0sp\_6sp\_9vr\_6sg\_4sp\_7sc\_1sp\_7sp\_9vr\_3sg\_4dc\_0sp\_9kw\_40vr\_7sg\_4dc\_1sp\_9kw\_41sp\_0kw\_46vr\_8sg\_4dc\_2sp\_9vr\_8sg\_2dc\_3sp\_9vr\_8sg\_0sg\_0sp\_1sp\_2kw\_44sp\_0vr\_8sg\_5sg\_4dc\_4sp\_1sp\_2vr\_9sg\_2sg\_2vr\_8sp\_9sp\_3sp\_3kw\_64dc\_5sp\_9sp\_3