**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**Лабораторная работа №6**

**по дисциплине «Языки программирования»**

Деревья

Группа: **АВТ-615**

Студент: **Кузенков В.С.**

Преподаватель: **Балагуров М.В.**

НОВОСИБИРСК 2017

# Постановка задачи

Выражение, содержащее целые константы, арифметические операции и скобки, может быть представлено в виде двоичного дерева. Концевая вершина дерева должна содержать значение константы. Промежуточная - код операции и указатели на правый и левый операнды - вершины дерева. Функция получает строку, содержащую выражение, и строит по ней дерево. Другая функция производит вычисления по полученному дереву.

# Ограничения

Отсутствуют

# Тестирование

**Входные данные**

**Выходные данные**

# 

# Текст программы:

#include <iostream>

using namespace std;

/\* Выражение, содержащее целые константы, арифметические операции и скобки, может быть представлено в виде двоичного дерева. Концевая вершина дерева должна содержать значение константы. Промежуточная - код операции и указатели на правый и левый операнды - вершины дерева. Функция получает строку, содержащую выражение, и строит по ней дерево. Другая функция производит вычисления по полученному дереву.\*/

#define LEN 64

struct node

{

char var[LEN];

node \*parent = NULL;

node \*child1 = NULL;

node \*child2 = NULL;

};

char\* fixStr(char\*);

node\* splitToTree(char\*,node\*);

int culcTree(node\*);

bool isDigit(char);

void printTree(node\*,int);

int main()

{

setlocale(0, "rus");

char str[64];

cout << "Введите выражение:" << endl;

cin >> str;

node \*root = new node;

root->var[0] = '\0';

fixStr(str);

splitToTree(str, root);

printTree(root,0);

cout << "Итог: " << culcTree(root) << endl;

system("pause");

return 0;

}

char \*fixStr(char str[64]) //хук на отрицательные числа

{

char newStr[64];

int j = 0;

bool flag = 0;//был ли вставлен 0-

for (int i = 0; str[i] != '\0'; ++i, ++j) //считаем вес всех операторов в строке

{

if(i == 0 && str[i] == '-')

{

newStr[0] = '(';

newStr[1] = '0';

j++;

j++;

newStr[j] = str[i];

flag = 1;

continue;

}

if(str[i] == '-' && !isDigit(str[i - 1]) && isDigit(str[i+1]))

{

newStr[j] = '(';

j++;

newStr[j] = '0';

j++;

flag = 1;

newStr[j] = str[i];

continue;

}

if (str[i] == '-' && isDigit(str[i + 1]))

{

newStr[j] = '+';

j++;

newStr[j] = '(';

j++;

newStr[j] = '0';

j++;

flag = 1;

newStr[j] = str[i];

continue;

}

if (flag) {

newStr[j++] = ')';

flag = 0;

}

newStr[j] = str[i];

}

newStr[j] = '\0';

for (int i = 0; newStr[i] != 0;i++) str[i] = newStr[i];

str[j] = '\0';

return str;

}

node\* splitToTree(char\* str, node\* root)

{

unsigned short operatorValue = 0;

int eachOpVal[LEN];

int bracketValue = 0; //вес скобок в скобках

for (int i = 0; i < LEN; ++i) eachOpVal[i] = -1;

//посчитаем определим максимальный вес операции

for (int i = 0; str[i] != '\0'; ++i) //считаем вес всех операторов в строке

{

if (isDigit(str[i]))

{

eachOpVal[i] = 0;

continue;

}

if (str[i] == '-' || str[i] == '+') {

eachOpVal[i] = 1 + bracketValue;

operatorValue += 1 + bracketValue;

continue;

}

if (str[i] == '\*' || str[i] == '/') {

eachOpVal[i] = 2 + bracketValue;

operatorValue += 2 + bracketValue;

continue;

}

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

eachOpVal[i] = 0;

bracketValue+=2;

continue;

}

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

{

eachOpVal[i] = 0;

bracketValue-=2;

}

}

//если операций не осталось, значит время вернуть корневую вершину

if (operatorValue == 0)

{

int strI = 0;

for (int i = 0; str[i] != 0; i++)

{

root->var[strI+1] = '\0';

if (str[i] == '(' || str[i] == ')') continue;

root->var[strI++] = str[i];

}

root->child1 = NULL;

root->child2 = NULL;

return root;

}

//ищем то, от куда будем начинать и создаем из этого вершину

int minValue = 0;

for (int i = 0; eachOpVal[i] != -1; i++)

{

if (eachOpVal[i] != 0) minValue = i; //первый встреченный оператор

}

for (int i = 0; eachOpVal[i] != -1; i++)

{

if (eachOpVal[minValue] > eachOpVal[i] && eachOpVal[i] != 0) minValue = i;

}

root->var[0] = str[minValue];

root->var[1] = '\0';

//делим строку относительно операции

char strRight[LEN];

char strLeft[LEN];

int strLeftI = 0;

int strRightI = 0;

for (int i = 0; i != minValue; i++) {

strLeft[strLeftI++] = str[i];

}

strLeft[strLeftI] = '\0';

for (int i = minValue+1; str[i] != 0; i++)

{

strRight[strRightI++] = str[i];

}

strRight[strRightI] = '\0';

node \*leftChild = new node;

leftChild->parent = root;

node \*rightChild = new node;

rightChild->parent = root;

root->child1 = leftChild;

root->child2 = rightChild;

splitToTree(strLeft, leftChild);

splitToTree(strRight, rightChild);

}

void printTree(node \*root,int level)

{

if (root)

{

printTree(root->child1, level + 1);

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

if(root->parent)

{

for (int j = 0; root->parent->var[j] != '\0'; ++j)

cout << " ";

}

}

cout << root->var << endl;

printTree(root->child2, level + 1);

}

}

int culcTree(node\* root)

{

if (root)

{

if (root->child1 == NULL && root->child2 == NULL && isDigit(root->var[0])) {

return atoi(root->var);

}

else

{

int v = culcTree(root->child1);

int w = culcTree(root->child2);

if (root->var[0] == '+')

return v + w;

if (root->var[0] == '-')

return v - w;

if (root->var[0] == '\*')

return v\*w;

if (root->var[0] == '/')

return v / w;

}

}

}

bool isDigit(char val)

{

return val >= '0' && val <= '9' ? 1 : 0;

}

**Вывод:** Определение дерева имеет рекурсивную природу. Элемент этой структуры данных называется **вершиной**. Дерево представляет собой вершину, имеющую ограниченное число связей (**ветвей**) к другим деревьям. Нижележащие деревья для текущей вершины называются **поддеревьями**, а их головные вершины - **потомками**. По отношению к потомкам текущая вершина называется **предком**. Вершины, не имеющие потомков, называются оконечными или **терминальными,**головная вершина всего дерева называется **корневой**.