Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное
учреждение высшего образования
Национальный исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчет по лабораторной работе

«Вычисление арифметических выражений»

Выполнил:
студент группы 382003-1
Маслов А.Е.

Проверил:
преподаватель каф. МОСТ,
Арисова А.Н.

Нижний Новгород
2021

Содержание

Постановка задачи

Разработать программу, реализующую шаблонный класс stack.

Разработать программу, выполняющую вычисление арифметического выражения с вещественными числами. Выражение в качестве операндов может содержать переменные и вещественные числа. Допустимые операции известны: +, -, /, *. Допускается наличие знака "-" в начале выражения или после открывающей скобки. Программа должна выполнять предварительную проверку корректности выражения и сообщать пользователю вид ошибки и номера символов строки, в которых были найдены ошибки.

Проверить работу программ с помощью Google test.

Метод решения

Класс stack реализует обычный стек (абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»)) с ограничением на максимальное количество элементов.

Класс myexpression хранит выражение с возможностью его проверки методом correct. Данный метод отслеживается ошибки в выражении и сообщает тип ошибки и номер символа, где она допущена.

Метод correct проверяет выражение на данные типы ошибок:

- Пустое выражение
- Неправильно расставленные скобки (Например: (a+9)(), ((4+2))),)3+5)
- Неправильное расположение точек (Например: 5.a+2, 5..2, 2.2.2)
- Имена переменных не содержат цифр (Например: 5a+2, a5A+2)
- Неправильное расположение операций (Например: 5+/2, *(a-5))
- Неизвестные символы (Например: 5+2%, a?-5)

Класс postfix_entry хранит корректное выражение в обратной польской записи. Класс имеет конструктор от класса myexpression, где если выражение корректно оно переводится в обратную польскую запись.

Алгоритм перевода:

- Если символ является числом или переменной, добавляем его к выходной строке.
- Если символ является унарным минусом, заменяем его на ~ и помещаем в стек.
- Если символ является открывающей скобкой, помещаем его в стек.
- Если символ является закрывающей скобкой:

До тех пор, пока верхним элементом стека не станет открывающая скобка, перемещаем элементы из стека в выходную строку. При этом открывающая скобка удаляется из стека, но в выходную строку не добавляется.

- Если символ является бинарной операцией, тогда:

Пока на вершине стека унарным минус или операция на вершине стека приоритетнее или с одинаковым приоритетом. Перемещаем верхний элемент стека в выходную строку. Затем помещаем операцию в стек.

- Когда входная строка закончилась, перемещаем все символы из стека в выходную строку.

В ходе работы алгоритма в выходной строке операции и операнды разделяются между собой пробелами.

Также класс postfix_entry имеет метод computation, который производит вычисление данного выражения.

Алгоритм вычисления:

- Если число, помещаем его в стек.
- Если переменная, просим ввести её значение. После ввода, помещаем в стек. Если пользователь не хочет вводить значение, то останавливаем вычисления и возвращаем 0.
- Если знак операции, то соответствующая операция выполняется над требуемым количеством значений, извлечённых из стека, взятых в порядке добавления. Результат выполненной операции кладётся на вершину стека.
- После вычислений результат лежит на вершине стека.

Руководство пользователя

Пользователя встречает предложение ввести выражение (без пробелов). Ввод должен быть корректен, иначе пользователя попросят ввести выражение снова.

Затем поступит предложение перевести выражение в обратную польскую запись. Если пользователь соглашается, то ему показывают вид выражения в обратной польской записи и предлагают провести вычисления. Если получаем согласие, то проводим вычисления, если в ходе вычислений встречается переменная, то просим пользователя ввести её. После ввода продолжаем вычисления и показываем результат.

Если пользователь захочет провести вычисления с новыми значениями переменных или проверить неизменность результата вычислений, то он может согласиться провести вычисление данного выражения ещё раз.

Если пользователь отказывается на одном из вышеперечисленных предложений, то ему предлагают закончить работу с программой или продолжить и ввести новое выражение.

Пример интерфейса на простом выражении:

```
Enter your expression (without spaces):
1+b
Translate the expression into reverse Polish notation?: 1 - Yes, 2 - No
1
Reverse Polish entry(^ - unary minus): 1 b +
Calculate the expression?: 1 - Yes, 2 - No
1
Do you want to enter the value of a variable b?: 1 - Yes, 2 - No
1
3.55
Result: 4.55
Calculate the expression again?: 1 - Yes, 2 - No
1
Do you want to enter the value of a variable b?: 1 - Yes, 2 - No
1
-6.7
Result: -5.7
Calculate the expression again?: 1 - Yes, 2 - No
2
Do you want to finish?: 1 - Yes, 2 - No
1
```

Описание программной реализации

В решение входят несколько проектов:

- arithmetic – содержит объявление и реализацию классов stack, myexpression и postfix_entry.
    - stack.h – реализация класса stack.
    - arithmetic.h – содержит объявление myexpression и postfix_entry.
    - arithmetic.cpp – содержит реализацию методов классов myexpression и postfix_entry.
- gtest - проект google тестов.
- sample – содержит реализацию пользовательского приложения для задания, перевода в обратную польскую запись и вычисления арифметических выражений.
    - main_arithmetic.cpp – содержит реализацию интерфейса пользователя.
- tests – проект с реализацией тестов для всех 3 классов.
    - test_arithmetic.cpp - тесты для myexpression и postfix_entry.
    - test_stack.cpp - тесты для stack.
    - test_main.cpp - запуск тестов.

Описание полей и методов:

- stack
    - size_t capacity - вместимость стека.

    - size_t size - количество элементов в стеке.
    - T* data - массив элементов типа T.
    - stack(const size_t capacity_ = 10) - конструктор стека в зависимости от ёмкости.
    - ~stack() - деструктор.
    - size_t stack_size() - возвращает текущие количество элементов в стеке.
    - bool empty() - проверка на пустоту.
    - bool full() - проверка на заполненность стека.
    - void clear() - очистка стека.
    - void change_capacity() - изменение вместимости стека.
    - void push(const T& elem) - вставка элемента.
    - T pop() - извлечение элемента.
    - T& top() - просмотр верхнего элемента стека без его удаления.
- myexpression
    - string exp - строка хранит выражение.
    - myexpression() - конструктор по умалчанию.
    - myexpression(const string& exp_) - конструктор преобразования от строки.
    - myexpression(const myexpression& other) - конструктор копирования.
    - bool correct() const - проверяет выражение на корректность.
    - size_t exp_size() const - возвращает длину выражения.
- postfix_entry
    - string pe - хранит выражение в обратной польской записи.
    - postfix_entry(const myexpression& exp_) - конструктор преобразования от myexpression. Преобразует выражение в обратную польскую запись.
    - double computation() - вычисляет выражение.
    - string get_postfix_entry() const - возвращает хранящееся выражение.

Подтверждение корректности

Корректность выражений проверялась с помощью google тестов. Результат прохождения тестов:

```
[----------] Global test environment tear-down
[==========] 24 tests from 3 test cases ran. (284 ms total)
[  PASSED  ] 24 tests.
```

Подробнее тесты можно посмотреть в файлах test_arithmetic.cpp и test_stack.cpp.

Также проверим корректность на достаточно сложном примере:

```
1.55+2*(-45.9/9)/1.275
Translate the expression into reverse Polish notation?: 1 - Yes, 2 - No
1
Reverse Polish entry(~ - unary minus): 1.55 2 45.9 ~ 9 / * 1.275 / +
Calculate the expression?: 1 - Yes, 2 - No
1
Result: -6.45
Calculate the expression again?: 1 - Yes, 2 - No
2
Do you want to finish?: 1 - Yes, 2 - No
1
```

Результат верен.

Заключение

Цель достигнута. Реализованы классы stack, myexpression и postfix_entry, а их корректность проверена с помощью google тестов.

Приложение

Код проекта размещён на git hub. Ради примера прилагаю реализацию метода вычисления выражения:

```
double postfix_entry::computation()
{
size_t dotflag = 0;
bool flagn = false;

stack <double> dst;
double temp;
string var;
char check;

for (size_t i = 0; i < pe.size(); i++)
{
if (letter(pe[i]))
{
var.push_back(pe[i]);
}
if (pe[i] == ' ' || (letter(pe[i]) && i + 1 == pe.size()))
{
dotflag = 0;
flagn = false;
if (var != "")
{
cout << "Do you want to enter the value of a variable " << var << "?: 1 - Yes, 2 - No\n";
cin >> check;
if (check == '1')
{
cin >> temp;
dst.push(temp);
}
```

```
else
{
cout << "Variables in expression\n";
return 0;
}
}
var.clear();
}
if (number(pe[i]))
{
if (dotflag == 0)
{
if (flagn)
{
temp = dst.pop() * 10 + double(pe[i]) - 48;
dst.push(temp);
}
else
{
dst.push(double(pe[i]) - 48);
}
flagn = true;
}
else
{
temp = dst.pop() + ((double(pe[i]) - 48) / (pow(10, dotflag)));
dst.push(temp);
dotflag++;
}
}
if (pe[i] == '.')
{
if ((i == 0 || !number(pe[i - 1])))
```

```
{
dst.push(0.0);
}
flagn = false;
dotflag++;
}

switch (pe[i])
{
case '~':
temp = -(dst.pop());
dst.push(temp);
break;
case '+':
temp = (dst.pop() + dst.pop());
dst.push(temp);
break;
case '-':
temp = (-dst.pop() + dst.pop());
dst.push(temp);
break;
case '*':
temp = (dst.pop() * dst.pop());
dst.push(temp);
break;
case '/':
temp = dst.pop();
if (temp == 0)
{
cout << "Error division by zero\n";
return 0;
}
temp = dst.pop() / temp;
```

```
dst.push(temp);
break;
default:
break;
}
}
return dst.pop();
}
```