

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №4
по дисциплине «Программирование»
Тема: Динамические структуры данных. C++ intro

Студент гр. 0382

Азаров М.С.

Преподаватель

Берленко Т.А.

Санкт-Петербург

2021

Цель работы.

Изучить принципы работы с динамическими структурами и ознакомиться с основами написания программы на языке C++.

Задание.

Стековая машина.

Требуется написать программу, которая последовательно выполняет подаваемые ей на вход арифметические операции над числами с помощью стека на базе массива.

1) Реализовать класс **CustomStack**, который будет содержать перечисленные ниже методы. Стек должен иметь возможность хранить и работать с типом данных `int`

Объявление класса стека:

```
class CustomStack {  
  
    public:  
        // методы push, pop, size, empty, top + конструкторы, деструктор  
  
    private:  
        // поля класса, к которым не должно быть доступа извне  
  
    protected: // в этом блоке должен быть указатель на массив данных  
        int* mData;  
};
```

Перечень методов класса стека, которые должны быть реализованы:

- `void push(int val)` - добавляет новый элемент в стек
- `void pop()` - удаляет из стека последний элемент
- `int top()` - доступ к верхнему элементу
- `size_t size()` - возвращает количество элементов в стеке

- bool **empty()** - проверяет отсутствие элементов в стеке
- void **extend**(int n) - расширяет исходный массив на n ячеек

2) Обеспечить в программе считывание из потока **stdin** последовательности (не более 100 элементов) из чисел и арифметических операций (+, -, *, / (деление нацело)) разделенных пробелом, которые программа должна интерпретировать и выполнить по следующим правилам:

- Если очередной элемент входной последовательности - число, то положить его в стек
- Если очередной элемент - знак операции, то применить эту операцию над двумя верхними элементами стека, а результат положить обратно в стек (следует считать, что левый операнд выражения лежит в стеке глубже)
- Если входная последовательность закончилась, то вывести результат (число в стеке)

Если в процессе вычисления возникает ошибка:

- например вызов метода pop или top при пустом стеке (для операции в стеке не хватает аргументов)
- по завершении работы программы в стеке более одного элемента

программа должна вывести "**error**" и завершиться.

Примечания:

1. Указатель на массив должен быть protected.
2. Подключать какие-то заголовочные файлы не требуется, всё необходимое подключено
3. Предполагается, что пространство имен std уже доступно
4. Использование ключевого слова using также не требуется

Пример

Исходная последовательность: 1 -10 - 2 *

Результат: 22

Основные теоретические положения

Стек - это структура данных, в которой хранятся элементы в виде последовательности, организованной по принципу LIFO (Last In — First Out). Такую структуру данных можно сравнить со стопкой тарелок или магазином автомата. Стек не предполагает прямого доступа к элементам и список основных операций ограничивается операциями помещения элемента в стек и извлечения элемента из стека. Их принято называть PUSH и POP соответственно. Также, обычно есть возможность посмотреть на верхний элемент стека не извлекая его (TOP) и несколько других функций, таких как проверка на пустоту стека и некоторые другие.

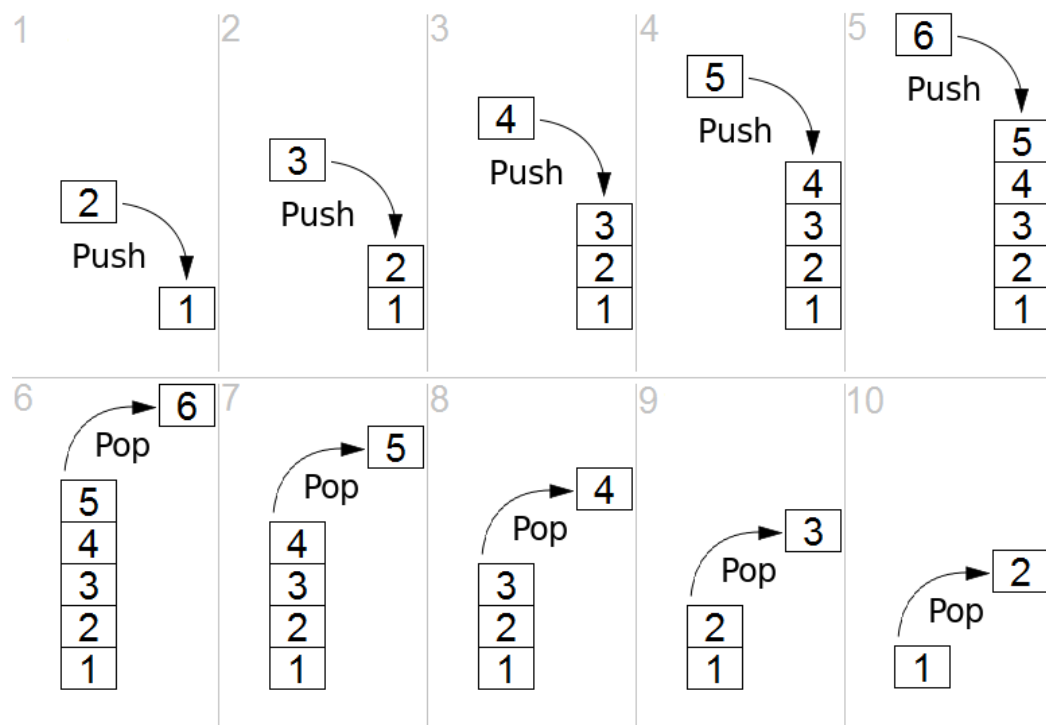


Рисунок 1: Пример добавления и удаления элементов из непустого стека

Стек можно легко реализовать на основе массива. Для этого достаточно хранить индекс "верхнего" элемента в стеке. Операция добавления сопровождается инкрементом этого индекса и записью в соответствующую ячейку нового значения. Операция извлечения сопровождается декрементом этого индекса. Дополнительно, может потребоваться реализовать возможность увеличения и уменьшения размера массива.

Классы

Проблема

В языке C есть возможность определять структуры, т.е. новые типы данных, которые являются композицией из уже существующих типов. Однако структура в C определяется только данными, например:

```
struct Point { // Структура в C это объединение различных типов
    данных в новый, единый тип данных
    int x;
    int y;
}
```

Язык C++ реализует объектно-ориентированную парадигму программирования, которая включает в себя реализацию механизма инкапсуляции данных. Инкапсуляция в C++ подразумевает, что:

1. В одной языковой конструкции размещаются как данные, так и функции для обработки этих данных

2. Доступ к данным извне этой конструкции ограничен, иными словами, напрямую редактировать данные как в структурах C нельзя. Пользователю предоставляется интерфейс из методов (API) с помощью которого он может влиять на состояние данных.

Структуры из C не подходят по обоим параметрам, язык C не поддерживает объектно-ориентированную парадигму.

P.S Причина ввода классов описанная выше - не единственная, но в рамках этого степа и курса её достаточно.

Решение

Для того, чтобы обеспечить такую инкапсуляцию данных, в C++ ввели классы. Класс - это шаблон, по которому определяется форма объекта. В нем указываются данные и код, который будет оперировать этими данными

По-другому, класс - это абстрактный тип данных, который может включать в себя не только данные, но и программный код в виде функций. Они реализуют в себе оба принципа, описанных выше следующим образом:

В классе могут размещаться как данные (их называют полями), так и функции (их называют методы) для обработки этих данных.

Любой метод или поле класса имеет свой спецификатор доступа: public, private или protected (его мы не будем рассматривать).

Приведём пример, как может выглядеть объявление (сигнатура) класса поезда:

```
class Train {  
public:  // публичные поля/методы класса.  
  
    // это конструктор. Здесь происходит начальная инициализация полей  
    класса  
    // конструктор вызывается всегда при создании нового экземпляра  
    класса (объекта)  
    // он может как принимать аргументы так и не принимать. Может быть  
    несколько конструкторов  
  
    Train() { // конструктор по-умолчанию, вызывается когда при создании  
    объекта явно не указывается  
        // тип конструктора  
        mWagonsCount = 0;  
    }
```

```

        mName = new char[10];
        strncpy(mName, "Thompson", 8);
};

Train(size_t start_count, char* name)
    : mWagonsCount(start_count) // Это список инициализации полей
класса,
// можно инициализировать сколь
угодно полей таким образом
{
    mName = new char[strlen(name)];
    strncpy(mName, name, strlen(name));
}

// это деструктор. Здесь обычно происходит освобождение памяти,
выделенной полям класса
// деструктор вызывается всегда при уничтожении экземпляра класса
// например, когда заканчивается область видимости переменной
~Train() {
    delete[] mName;
};

// прочие методы класса, их можно вызывать извне, имея экземпляр
класса Train
void pushWagons(size_t count) {
    if(mWagonsCount + count < 15)
        mWagonsCount+=count;
}
size_t wagonsCount() {
    return mWagonsCount;
}

private: // приватные поля/методы класса, недоступные пользователю,
работающему с классом
    size_t mWagonsCount;
    char* mName;

```

```
};
```

Train - это новый тип данных у каждого объекта которого будут свои значения полей и к каждому объекту привязаны методы, которые будут взаимодействовать ТОЛЬКО с данными объекта для которого они вызваны

Класс и методы описаны. Приведём пример, как создавать экземпляры класса и взаимодействовать с ними

```
int main()
{
    {
        Train loko1;

        char name[] = "Lutik";
        Train loko2(5, name);

        loko1.mWagonsCount = 8; // Не работает, т.к поле приватное
        loko1.pushWagons(7); // прибавит 7 вагонов к поезду loko1

        size_t count = loko2.wagonsCount(); // помещает в переменную
count количество вагонов в поезде loko2
    } // Здесь кончается область видимости созданных внутри объектов и
вызовятся деструкторы для loko1 и loko2

    return 0;
}
```

Здесь видно, как работают принципы инкапсуляции. Пользователь не может напрямую изменить количество вагонов в конкретном поезде, у него для этого есть метод *pushWagons()*. Однако, с помощью этого метода не получится сделать вагонов больше 15, поскольку программист позаботился о том, чтобы поезд мог сдвинуться с места и никто не прикрепил к нему лишних вагонов.

Выполнение работы.

Класс *CustomStack*:

Поля класса:

- *int m_size* — количество переменных хранящихся в данный момент в стеке (*m_size-1* также является индексом последнего добавленного элемента).
- *int m_max_size* — максимально доступный размер стека в данный момент.
- *int* mData* — указатель на массив данных.

Методы класса :

- *CustomStack()* - конструктор класса в нем выделяется начальная память для массива *mData*.
- *~CustomStack()* - деструктор класса в нем очищается дин. память выделенная для массива *mData*.
- *void extend(int n)* - расширяет исходный массив на *n* ячеек .
- *bool empty()* - проверяет отсутствие элементов в стеке.
- *int size()* - возвращает количество элементов в стеке.
- *int top()* - возвращает значение верхнего элемента.
- *int pop()* - возвращает значение верхнего элемента и удаляет его из стека.
- *void push(int val)* - добавляет новый элемент в стек.

Функция **try stoi(const string &s, int &i)**:

Описание:

Функция проверяет можно ли строку **s** привести к целочисленному значению. Если да то присваивает это значение в **i** и возвращает **true**, если нельзя привести, возвращает **false**.

Главная функция **main()**:

Описание:

Главная функция выполняющая поставленную задачу.

Ход работы:

- Считываем строки данных разделенные пробелом в **token**, пока не закончится ввод.
- Проверяем для каждого **token** можно ли привести его к **int**.
- Если можно привести **token** к **int** то добавляем это значение в стек **my_stack**.
- Если нет, тогда значит **token** хранит в себе арифметическое действие («+», «-», «\», «*»). Пытаемся достать из **my_stack** два верхних значения и сохранить их в **left_num** и **right_num**. При неудаче обрабатываем исключение. При удаче сохраняем в стек **my_stack** результат применения соответствующего арф. Действия.
- В итоге если в стеке остался один элемент то выводим его , если не один выводим ошибку.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	1 -10 - 2 *	22	Программа работает правильно
2.	1 2 + 3 4 - 5 * +	-2	Программа работает правильно

Выводы.

Была изучены принципы создания динамических структур и работы с ними. Также были изучены основы написания программы на языке C++.

Разработана программа, полностью выполняющая поставленную задачу, а именно последовательно выполняет подаваемые ей на вход арифметические операции над числами с помощью стека на базе массива . Для решения этой задачи были использованы полученные знания о том как устроен стек и как его можно реализовать на основе массивов. Также были применены полученные знания об ООП языка C++.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab_4.c

```
#include <cstring>
#include <iostream>

using namespace std;

class CustomStack {

public:
    CustomStack() {
        mData = new int[m_max_size];
    }

    ~CustomStack() {
        delete[] mData;
    }

    void extend(int n) {
        m_max_size += n;
        int *new_Data = new int[m_max_size];
        memcpy(new_Data, mData, m_size * sizeof(int));
        delete[] mData;
        mData = new_Data;
    }

    void push(int a) {
        if (m_size + 1 >= m_max_size) {
            extend(10);
        }
        mData[m_size] = a;
        m_size++;
    }
}
```

```

int pop() {
    if (empty()) {
        throw "error";
    }
    m_size--;
    return mData[m_size];
}

bool empty() {
    if (m_size == 0) {
        return true;
    }
    else {
        return false;
    }
}

int top() {
    if (empty()) {
        throw "error";
    }
    return mData[m_size-1];
}

int size() {
    return m_size;
}

private:
    int m_size = 0;
    int m_max_size = 100;

protected:
    int* mData;
};

```

```

bool try_stoi(const string &s, int &i){
    try {
        i = stoi(s);
        return true;
    }
    catch (const std::invalid_argument&) {
        return false;
    }
}

```

```

int main() {
    CustomStack my_stack;
    string token ;
    int value;
    int left_num , right_num;

    while (cin >> token) {
        if (try_stoi(token,value)) {
            my_stack.push(value);
        } else {
            try{
                right_num = my_stack.pop();
                left_num = my_stack.pop();
            }
            catch(const char* error_str) {
                cout << error_str;
                return 0;
            }

            if (token == "+") {
                my_stack.push(left_num + right_num);
            }
            if (token == "-") {

```

```

        my_stack.push(left_num - right_num);
    }
    if (token == "*") {
        my_stack.push(left_num * right_num);
    }

    if (token == "/") {
        my_stack.push(left_num / right_num);
    }
}

if (my_stack.size() != 1) {
    cout << "error";
    return 0;

} else {
    cout << my_stack.pop();
    return 0;
}

}

```