

# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 3](#_Toc73903720)

[ВВЕДЕНИЕ 4](#_Toc73903721)

[ОСНОВНАЯ ЧАСТЬ 5](#_Toc73903722)

[Постановка задачи 5](#_Toc73903723)

[Описание входных данных 7](#_Toc73903724)

[Описание выходных данных 7](#_Toc73903725)

[Метод решения 8](#_Toc73903726)

[Описание алгоритма 11](#_Toc73903727)

[Блок-схема алгоритма 25](#_Toc73903728)

[Код программы 35](#_Toc73903729)

[Тестирование 35](#_Toc73903730)

[ЗАКЛЮЧЕНИЕ 36](#_Toc73903731)

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ (источников) 37](#_Toc73903732)

[ПРИЛОЖЕНИЕ А 38](#_Toc73903733)

# ВВЕДЕНИЕ

Объекто-ориентированное программирование - парадигма написания программ, пришедшая на замену функциональному подходу и представляющая из себя декомпозицию задачи на отдельные объекты, между которыми и происходит взаимодействие [5]. Преимущество ООП очевидно при необходимости программирования больших систем. Первоначально, когда обширные проекты пытались создавать императивным подходом, то код разрастался до совершенно не поддерживаемых габаритов. В случае появления ошибки, её невозможно было исправить, так как в такой системе весь код представляет из себя единое целое, и сбой в одной строчке по принципу домино ломает всю программу. ООП позволяет решить эту проблему и обеспечить надежную разработку и поддержку программ благодаря трём основным принципам - инкапсуляции, наследованию и полиморфизму. Инкапсуляция отделяет свойства и методы одного объекта от другого и декларирует, что всякое взаимодействие между ними возможно только по заранее созданным методам. Такой подход гарантирует безопасность, в случае сбоя в одном из объектов, это никак не повредит второй объект, так проще локализовать баг и исправить его. Наследование же оберегает программиста от написания одного и того же кода при описании свойств и методов нескольких похожих классов, вместо этого они объявляются родственными и каждый дочерний класс наследует, и соответсвенно может использовать, те или иные методы родительского класса. Это делает код чище и избавляет от повторений. Полиморфизм в свою очередь обеспечивает возможность функциям обрабатывать данные разных типов и в сочетании с наследованием дает мощнейщий инструмент создания гибких программ со сложной логикой при наименьшем количестве кода. По всем этим причинам в данной работе используется именно ООП подход, что в итоге позволило разработать эффективную и устойчивую программу.

# ОСНОВНАЯ ЧАСТЬ

## Постановка задачи

Разработать программу, в которой на вход подается целочисленное арифметическое выражение без скобок. Все операции имеют одинаковый приоритет. В выражении используются следующие операции: +, -, \* и % (целочисленный остаток от деления). После выполнения каждой операции в составе выражения выдается промежуточный результат. Работа завершается после выполнения последней операции.

**Пример**.

Исходное целочисленное арифметическое выражение: 5 + 6 – 1 \* 3  
Форма вывода промежуточных результатов:

5 + 6 = 11

11 – 1 = 10

10 \* 3 = 30

Использовать объекты:

1. Для ввода исходного целочисленного арифметического выражения.
2. Для отработки операции.
3. Для вывода очередного промежуточного результата.

Написать программу, реализующую следующий алгоритм:

1. Ввод арифметического выражения.
2. Определение первого операнда.
3. Выдача «signal\_1» и передача первого операнда.
4. Цикл до конца арифметического выражения
   1. Определение очередной операции.
   2. Выдача «signal\_2» и передача символа операции.
   3. Определение очередного операнда.
   4. Выдача «signal\_3» и передача очередного операнда.
5. Конец цикла.

В обработчиках объекта отработки операции реализовать:

При получении «signal\_1»: сохранить значение первого операнда.   
При получении «signal\_2»: сохранить символ операции.

При получении «signal\_3»: выполнить операцию, выдать «signal\_4» и передать строку для вывода.

## Описание входных данных

Целочисленное арифметическое выражение. Операнды и знаки операций разделены пробелом.

## Описание выходных данных

В первой строке:

«первый операнд» «знак первой операции» «второй операнд» = «результат»

В последующих строках:

«результат предыдущей операции» «знак очередной операции» «очередной операнд» = «результат»

Разделитель один пробел.

## Метод решения

Для решения задачи воспользуемся объектом app класса Application; объектами стандартного потока ввода и вывода - cin, cout; стандартными функциями getline, to\_string, stoi, pow. Все вводы происходят с клавиатуры, все выводы - на экран. Структура иерархии наследования классов представлена в таблице 1.

Таблица 1 - Иерархия наследования классов

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **№** | **Имя класса** | **Классы-наследники** | **Модификатор**  **доступа при наследовании** | **Описание** | **Номер** | **Комментарий** |
| 1 | Base |  |  | Базовый класс в иерархии классов. Содержит основные поля и методы. |  |  |
|  | Application | public |  | 2 |  |
|  | Operation\_handler | public |  | 3 |  |
|  | Output\_device | public |  | 4 |  |
| 2 | Application |  |  | Класс корневого объекта (приложения). |  |  |
| 3 | Operation\_ handler |  |  | Класс объектов, подчинённых корневому объекту класса Application. |  | Обеспечивает обработку необходимой операци в соответствии с посланным от корневого объекта сигналом |
| 4 | Output\_ dev ice |  |  | Класс объектов, подчинённых корневому объекту класса Application. |  | Обеспечивает вывод промежуточного результата на экран в соответсвии с посланным от объект отработки операции сигналом |

**Класс Base**

Методы:

**—** segmentation - метод расчленения исходного выражения на отдельные операнды и символы операций, разделенные пробелами.

**Класс Application**

Методы:

**—** buildTree - метод построения дерева иерархии (создан с опорой на учебный материал [6]).

**—** execute - метод отработки программы

**—** signal\_1 - метод сигнала, который используется для установки связи с методом обработчика handler\_1 объекта класса Operation\_handler (концепция сигналов и обработчиков изложена в пособии [3]).

**—** signal\_2 - метод сигнала, который используется для установки связи с методом обработчика handler\_2 объекта класса Operation\_handler.

**—** signal\_3 - метод сигнала, который используется для установки связи с методом обработчика handler\_3 объекта класса Operation\_handler.

**Класс Operation\_handler**

Поля:

**—** operation\_symbol - символ очередной операции. Тип - символьный. Модификатор доступа - private.

**—** first\_operand - очередной первый операнд. Тип - строковый. Модификатор доступа - private.

**—** second\_operand - очередной второй операнд. Тип - строковый. Модификатор доступа - private.

**—** temp\_result - промежуточный результат вычислений. Тип - строковый. Модификатор доступа - private.

Методы:

**—** Operation\_handler - параметризованный конструктор класса Operation\_handler

**—** handler\_1 - метод обработчика, сохранящий значение первого операнда.

**—** handler\_2 - метод обработчика, сохраняющий символ операции.

**—** handler\_3 - метод обработчика, выполняющий операцию, выдающий метод сигнала signal\_4 и передающий строку для вывода.

**—** signal\_4 - метод сигнала, который используется для установки связи с методом обработчика handler\_4 объекта класса Output\_device

**Класс Output\_device**

Поля:

**—** line\_break - логический флаг, который обеспечивает переход на новую строку только для второго и всех последующих выводов. Тип - логический, Модификатор доступа - private. По умолчанию имеет значение логическая ложь.

Методы:

**—** Output\_device - параметризованный конструктор класса Output\_device.

**—** handler\_4 - метод обработчика, выводящий результат промежуточных вычислений на экран.

## Описание алгоритма

Функция: main

Функционал: основной код программы

Параметры: без параметров

Возвращаемое значение: целое - индикатор корректной работы программы

Алгоритм метода представлен в таблице 2.

Таблица 2 – Основной код программы.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 |  | Создание объекта app класса Application с помощью конструктора, аргументы не передаются | 2 |  |
| 2 |  | Вызов метода buildTree объекта app | 3 |  |
| 3 |  | Вызов метода execute объекта app | 4 |  |
| 4 |  | Возврат нуля | ∅ |  |

Класс объекта: Base

Модификатор доступа: protected

Метод: segmentation

Функционал: метод расчленения исходного выражения на отдельные операнды и символы операций, разделенные пробелами.

Параметры: ссылка на строку s

Возвращаемое значение: тип с пустым множеством значением

Алгоритм метода представлен в таблице 3.

Таблица 3 – Метод расчленения выражения на операнды и символы.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 | Передана пустая строка | Возврат "" | ∅ | if (s == "") |
|  |  | 2 |  |
| 2 |  | объявление строковой переменной result | 3 |  |
| 3 |  | объявление целочисленной переменной difference и инициализация единицей | 4 | Чтобы отрезать строку вместе со следующим пробелом |
| 4 |  | объявление константной ссылки на переменную start\_delimiter строкового типа и инициализация "" | 5 |  |
| 5 |  | объявление константной ссылки на переменную stop\_delimiter строкового типа и инициализация " " | 6 |  |
| 6 |  | объявление беззнаковой переменной first\_delimiter\_pos и инициализация значением, возвращенным вызовом метода find через ссылку s с параметром start delimiter | 7 | Ищем начало отсечения. По сути можно не делать, т.к. в каждом случае это нулевой индекс, но пусть |
| Продолжение таблицы 3. | | |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 7 |  | объявление беззнаковой переменной last\_delimiter\_pos и инициализация значением,  возвращенным вызовом метода find\_first\_of через ссылку s с параметром stop\_delimiter, first\_delimiter\_pos | 8 | ищем конец отсечения - первая встреча разделителя,  следующего за индексом first\_delimiter\_pos |
| 8 | В строке остался только последний операнд | присваивание полю last\_delimiter значения, возвращенного вызовом метода find\_last\_of через ссылку s с параметром в виде значения, возвращенного методом back через ссылку s, + 1 | 9 | if (last\_delimiter\_pos == pow(2, 32) - 1) |
|  |  | 10 |  |
| 9 |  | присваивание значению difference ноль | 10 | в этом случае далее пробела нет |
| 10 |  | присваивание значению переменной result значения,возвращенного вызовом метода substr  через ссылку s с параметрами first\_delimiter\_pos, last\_delimiter\_pos - first\_delimiter\_pos | 11 |  |
| 11 |  | присваивание значению переданной по ссылке строки s значения, возвращенного вызовом метода substr через ссылку s c параметром last\_delimiter\_pos + difference | 12 |  |
| 12 |  | Возврат строкового значения переменной result | ∅ |  |

Класс объекта: Application

Модификатор доступа: public

Метод: buildTree

Функционал: метод построения дерева иерархии

Параметры: без параметров

Возвращаемое значение: тип с пустым множеством значений

Алгоритм метода представлен в таблице 4.

Таблица 4 – Алгоритм метода построения дерева.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Коммен- тарий** |
| 1 |  | Вызов метода SetName текущего объекта с параметром "app" | 2 |  |
| 2 |  | Создание объекта класса Operation\_handler с помощью конструктора, в качестве  параметров - указатель на текущий объект this, строка "obj\_op\_handler", целочисленное число 2, целочисленное число 1. Адрес объект присваиваем указателю obj\_op\_handler класса Base | 3 |  |
| 3 |  | Создание объекта класса Output\_device с помощью конструктора, в качестве параметров - указатель на текущий объект this, строка "obj\_out", целочисленное число 3, целочисленное число 1. Адрес объект присваиваем указателю obj\_out класса Base | 4 |  |
| 4 |  | Вызов метода set\_connection через указатель this с параметрами: ссылка на метод сигнала signal\_1 из области видимости класса Application, преобразованная с помощью макроопределения SIGNAL\_D, obj\_op\_handler, ссылка на метод обработчика handler\_1 из области видимости класса Operation\_handler, преобразованная с помощью макроопределения HANDLER\_D | 5 |  |
| Продолжение таблицы 4. | | |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Коммен- тарий** |
| 5 |  | Вызов метода set\_connection через указатель this с параметрами: ссылка на метод сигнала signal\_2 из области видимости класса Application, преобразованная с помощью макроопределения SIGNAL\_D, obj\_op\_handler, ссылка на метод обработчика handler\_2 из области видимости класса Operation\_handler, преобразованная с помощью макроопределения HANDLER\_D | 6 |  |
| 6 |  | Вызов метода set\_connection через указатель this с параметрами: ссылка на метод сигнала signal\_3 из области видимости класса Application, преобразованная с помощью макроопределения SIGNAL\_D, obj\_op\_handler, ссылка на метод обработчика handler\_3 из области видимости класса Operation\_handler, преобразованная с помощью макроопределения HANDLER\_D | 7 |  |
| 7 |  | Вызов метода set\_connection через указатель obj\_op\_handler с параметрами: ссылка на метод сигнала signal\_4 из области видимости класса Operation\_handler, преобразованная с помощью макроопределения SIGNAL\_D, obj\_out, ссылка на метод обработчика handler\_4 из области видимости класса Output\_device, преобразованная с помощью макроопределения HANDLER\_D | ∅ |  |

Класс объекта: Application

Модификатор доступа: public

Метод: signal\_1

Функционал: метод сигнала, который используется для установки связи с методом обработчика handler\_1 объекта класса Operation\_handler

Параметры: ссылка на строку nothing

Возвращаемое значение: тип с пустым множеством значений

Алгоритм метода представлен в таблице 5.

Таблица 5 – Алгоритм метода первого сигнала.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 |  |  | ∅ |  |

Класс объекта: Application

Модификатор доступа: public

Метод: signal\_2

Функционал: метод сигнала, который используется для установки связи с методом обработчика handler\_2 объекта класса Operation\_handler

Параметры: ссылка на строку nothing

Возвращаемое значение: тип с пустым множеством значений

Алгоритм метода представлен в таблице 6.

Таблица 6 – Алгоритм метода второго сигнала.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 |  |  | ∅ |  |

Класс объекта: Application

Модификатор доступа: public

Метод: signal\_3

Функционал: метод сигнала, который используется для установки связи с методом обработчика handler\_3 объекта класса Operation\_handler

Параметры: ссылка на строку nothing

Возвращаемое значение: тип с пустым множеством значений

Алгоритм метода представлен в таблице 7.

Таблица 7 – Алгоритм метода третьего сигнала.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 |  |  | ∅ |  |

Класс объекта: Operation\_handler

Модификатор доступа: public

Метод: Operation\_handler

Функционал: Параметризованный конструктор класса Operation\_handler

Параметры: parent - указатель на головной объект класса Base, строковое name - наименование текущего объекта, целочисленное class\_number - номер класса наследника, целочисленное state\_number - число, характеризующeе готовность объекта к работе

Возвращаемое значение: не возвращает значения

Алгоритм метода представлен в таблице 8.

Таблица 8 – Алгоритм конструктора класса Operation\_handler.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 |  | Вызов конструктора базового класса Base с параметрами parent, name, class\_number, state\_number | ∅ |  |

Класс объекта: Operation\_handler

Модификатор доступа: public

Метод: handler\_1

Функционал: метод обработчика, сохранящий значение первого операнда. Параметры: строка first\_operand

Возвращаемое значение: тип с пустым множеством значений

Алгоритм метода представлен в таблице 9.

Таблица 9 – Алгоритм метода первого обработчика.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Коммен- тарий** |
| 1 |  | Присваивание значению поля first\_operand текущего объекта значение параметра first\_operand | 2 |  |
| 2 |  | Присваивание значению поля temp\_result текущего объекта значение параметра first\_operand | ∅ |  |

Класс объекта: Operation\_handler

Модификатор доступа: public

Метод: handler\_2

Функционал: метод обработчика, сохраняющий символ операции.

Параметры: строка symbol

Возвращаемое значение: тип с пустым множеством значений

Алгоритм метода представлен в таблице 10.

Таблица 10 – Алгоритм метода второго обработчика.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 |  | Присваивание значению поля operation\_symbol текущего объекта значение первого элемента строки symbol | ∅ | Symbol имеет тип string (что по сути массив из char), operation\_symbol - char, все символы операций состоят из одного символа, поэтому это допустимое решение |

Класс объекта: Operation\_handler

Модификатор доступа: public

Метод: handler\_3

Функционал: метод обработчика, выполняющий операцию, выдающий метод сигнала signal\_4 и передающий строку для вывода.

Параметры: строка second\_operand

Возвращаемое значение: тип с пустым множеством значений

Алгоритм метода представлен в таблице 11.

Таблица 11 – Алгоритм метода третьего обработчика.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 |  | Присваивание значению поля second\_operand текущего объекта значение параметра second\_operand | 2 |  |
| 2 |  | Присваивание значению поля first\_operand текущего объекта значение поля temp\_result текущего объекта | 3 |  |
| 3 | Был введен символ операции  + | Присваивание значению поля temp\_result текущего объекта преобразованного к типу string с помощью функции to\_string значения сумммы значений поля temp\_result, преобразованного к типу int с помощью функции stoi и параметра second\_operand, преобразованного к типу int с помощью функции stoi | 4 |  |
| Был введен символ операции  - | Присваивание значению поля temp\_result текущего объекта преобразованного к типу string с помощью функции to\_string значения разности значений поля temp\_result, преобразованного к типу int с помощью функции stoi и параметра second\_operand,  преобразованного к типу int с помощью функции stoi | 4 |  |
| Продолжение таблицы 11 | | | | |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
|  | Был введен символ операции  \* | Присваивание значению поля temp\_result текущего объекта преобразованного к типу string с помощью функции to\_string значения произведения значений поля temp\_result, преобразованного к типу int с помощью функции stoi и параметра second\_operand, преобразованного к типу int с помощью функции stoi | 4 |  |
| Был введен символ операции  % | Присваивание значению поля temp\_result текущего объекта преобразованного к типу string с помощью функции to\_string значения целочисленного остатка от деления значений поля temp\_result, преобразованного к типу int с помощью функции stoi и параметра second\_operand, преобразованного к типу int с помощью функции stoi | 4 |  |
|  |  | 4 | В остальных случаях |
| 4 |  | Объявление строки output и инициализация значением: first\_operand + " " + operation\_symbol + " " + second\_operand + " = " + temp\_result | 5 |  |
| 5 |  | Вызов метода emit\_signal текущего объекта с параметрами ссылка на метод сигнала signal\_4 из области видимости класса Operation\_handler, преобразованная с помощью макроопределения SIGNAL\_D, строка output | ∅ |  |

Класс объекта: Operation\_handler

Модификатор доступа: public

Метод: signal\_4

Функционал: метод сигнала, который используется для установки связи с методом обработчика handler\_4 объекта класса Output\_device

Параметры: ссылка на строку output

Возвращаемое значение: тип с пустым множеством значений

Алгоритм метода представлен в таблице 12.

Таблица 12 – Алгоритм метода четвертого сигнала.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 |  |  | ∅ |  |

Класс объекта: Output\_device

Модификатор доступа: public

Метод: Output\_device

Функционал: Параметризованный конструктор класса Output\_device

Параметры: parent - указатель на головной объект класса Base, строковое name - наименование текущего объекта, целочисленное class\_number - номер класса наследника, целочисленное state\_number - число, характеризующeе готовность объекта к работе

Возвращаемое значение: не возвращает значения

Алгоритм метода представлен в таблице 13.

Таблица 13 – Алгоритм метода вывода промежуточного результата.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 |  | Вызов конструктора базового класса Base с параметрами parent, name, class\_number, state\_number | ∅ |  |

Класс объекта: Output\_device

Модификатор доступа: public

Метод: handler\_4

Функционал: метод обработчика, выводящий результат промежуточных вычислений на экран

Параметры: строка temp\_result

Возвращаемое значение: тип с пустым множеством значений

Алгоритм метода представлен в таблице 14.

Таблица 14 – Алгоритм метода четвертого обработчика.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 | Это НЕ первый вывод промежутчного результата | Переход на новую строку | 2 | if (line\_break) |
|  |  | 2 |  |
| 2 |  | Вывод параметра temp\_result | 3 |  |
| 3 |  | Присваивание полю line\_break текущего объекта значения true | ∅ |  |

Класс объекта: Application

Модификатор доступа: public

Метод: execute

Функционал: метод отработки программы

Параметры: без параметров

Возвращаемое значение: тип с пустым множеством значений

Алгоритм метода представлен в таблице 15.

Таблица 15 – Алгоритм метода отработки программы.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 1 |  | Объявление строковых переменных number1, number2, symbol | 2 |  |
| 2 |  | Объявление строки expression | 3 |  |
| 3 |  | Вызов функции getline с параметрами cin, expression | 4 |  |
| 4 |  | Присваивание значению number1 результата, возвращенного вызовом функции segmentation текущего объекта с параметром expression | 5 |  |
| 5 |  | Вызов метода emit\_signal текущего объекта с параметрами: ссылка на метод сигнала signal\_1 из области видимости класса Application, преобразованная с помощью макроопределения SIGNAL\_D, строка number1 | 6 |  |
| 6 | Арифметическое выражение НЕ посчитано до конца |  | 7 | while (expression != "") |
|  |  | ∅ |  |
| 7 |  | Присваивание значению symbol результата, возвращенного вызовом функции segmentation текущего объекта с параметром expression | 8 |  |
| Продолжение таблицы 15. | | |
| **№** | **Предикат** | **Действия** | **№ перехода** | **Комментарий** |
| 8 |  | Вызов метода emit\_signal текущего объекта с параметрами: ссылка на метод сигнала signal\_2 из области видимости класса Application, преобразованная с помощью макроопределения SIGNAL\_D, строка symbol | 9 |  |
| 9 |  | Присваивание значению number2 результата, возвращенного вызовом функции segmentation текущего объекта с параметром expression | 10 |  |
| 10 |  | Вызов метода emit\_signal текущего объекта с параметрами: ссылка на метод сигнала signal\_3 из области видимости класса Application, преобразованная с помощью макроопределения SIGNAL\_D, строка number2 | 6 |  |

## Блок-схема алгоритма

Блок-схемы алгоритмов методов представлены на рисунках 1-10.

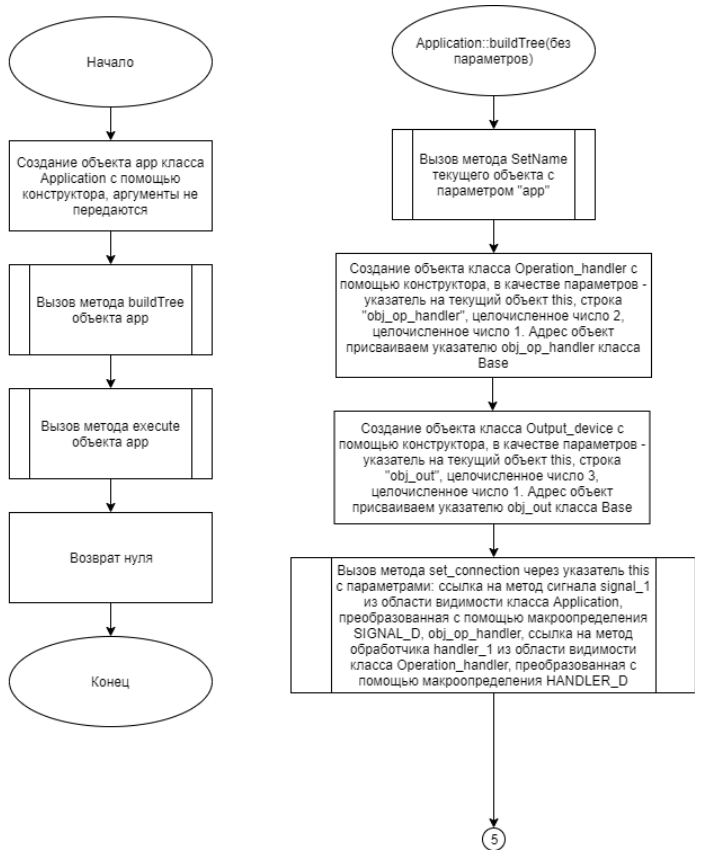


Рисунок 1 — Блок-схема алгоритма, часть 1

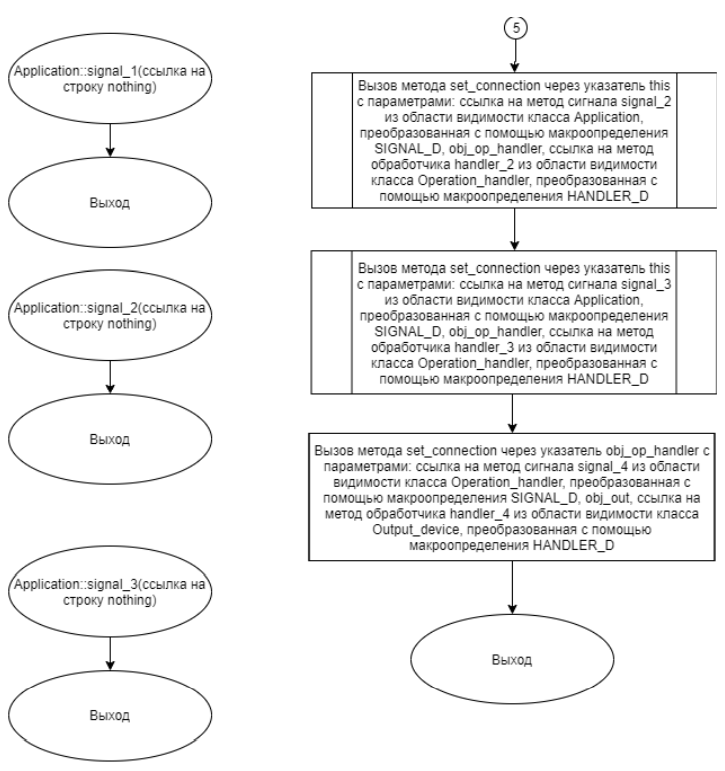


Рисунок 2 — Блок-схема алгоритма, часть 2

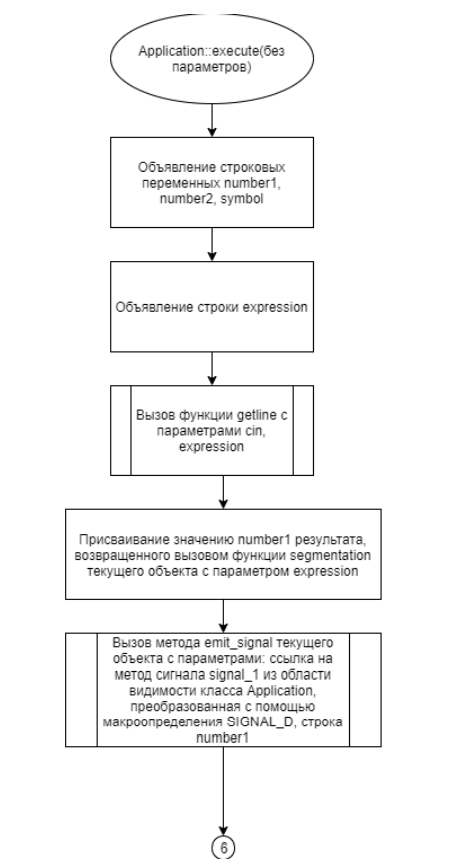


Рисунок 3 — Блок-схема алгоритма, часть 3

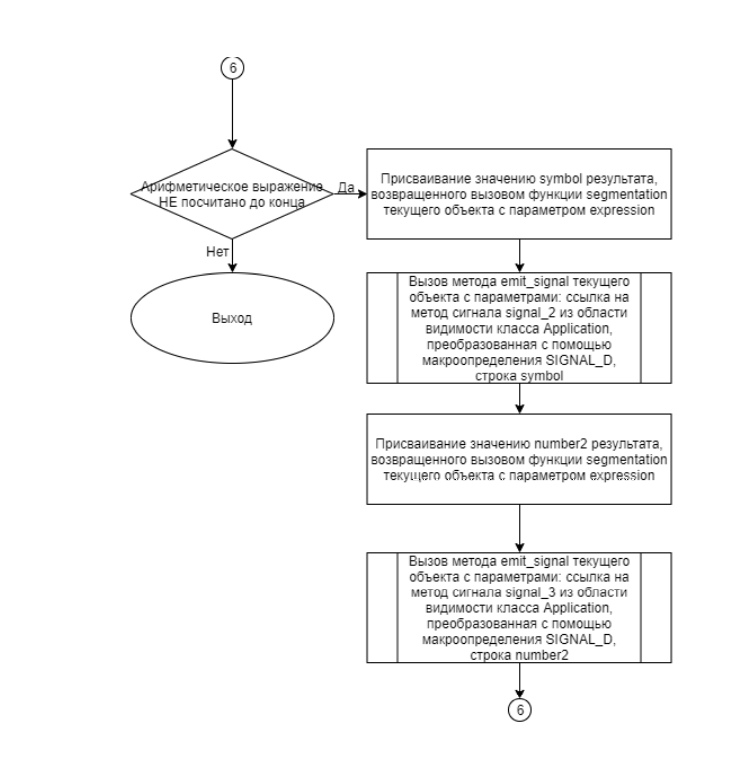


Рисунок 4 — Блок-схема алгоритма, часть 4

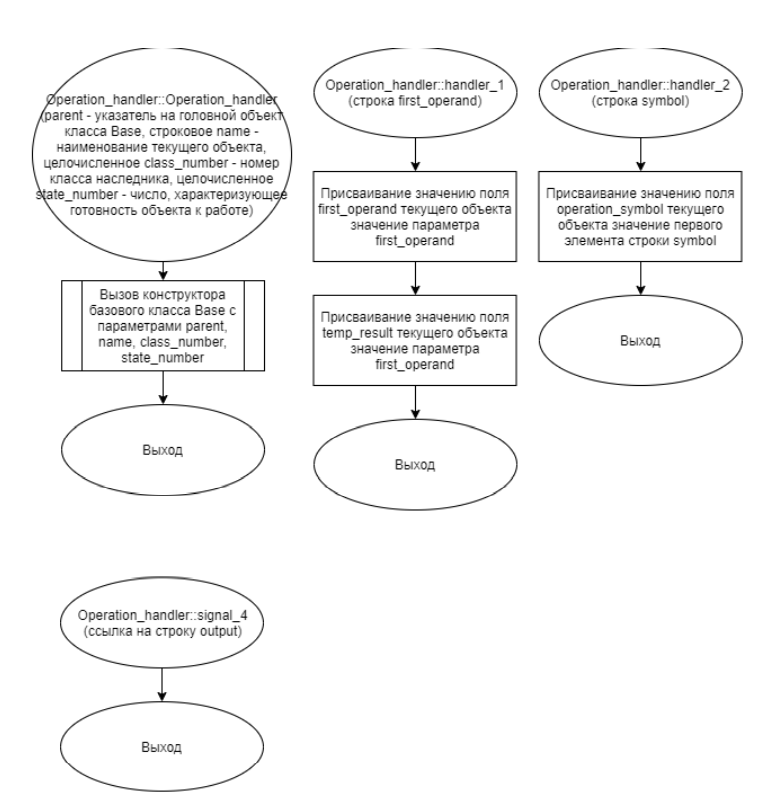


Рисунок 5 — Блок-схема алгоритма, часть 5

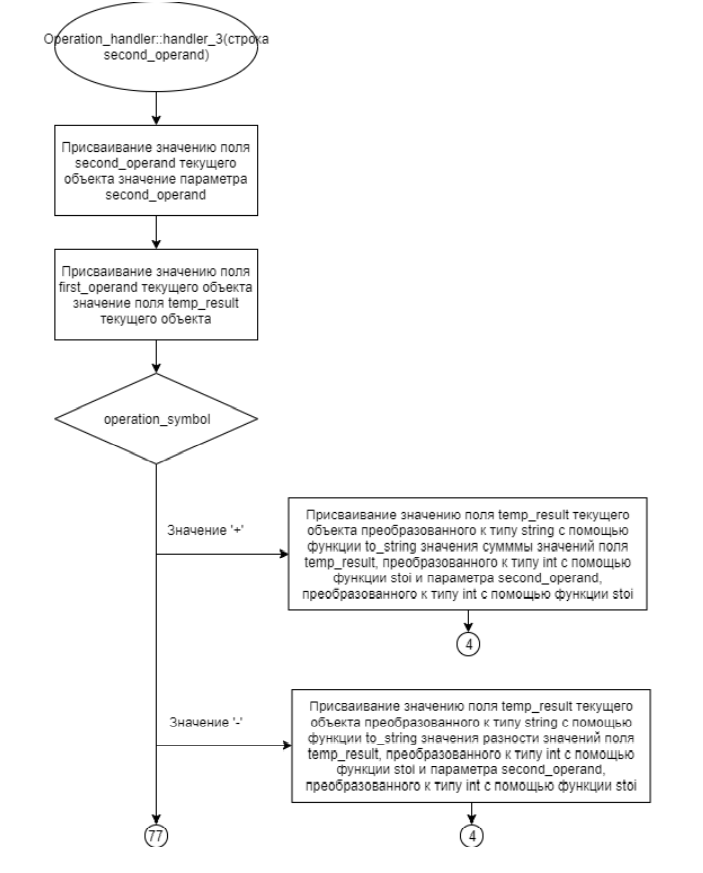


Рисунок 6 — Блок-схема алгоритма, часть 6

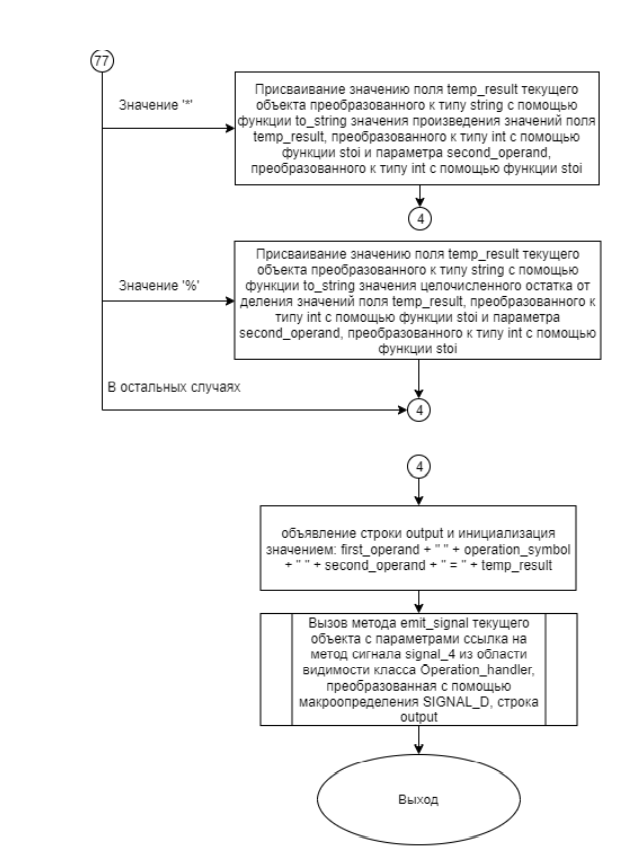


Рисунок 7 — Блок-схема алгоритма, часть 7

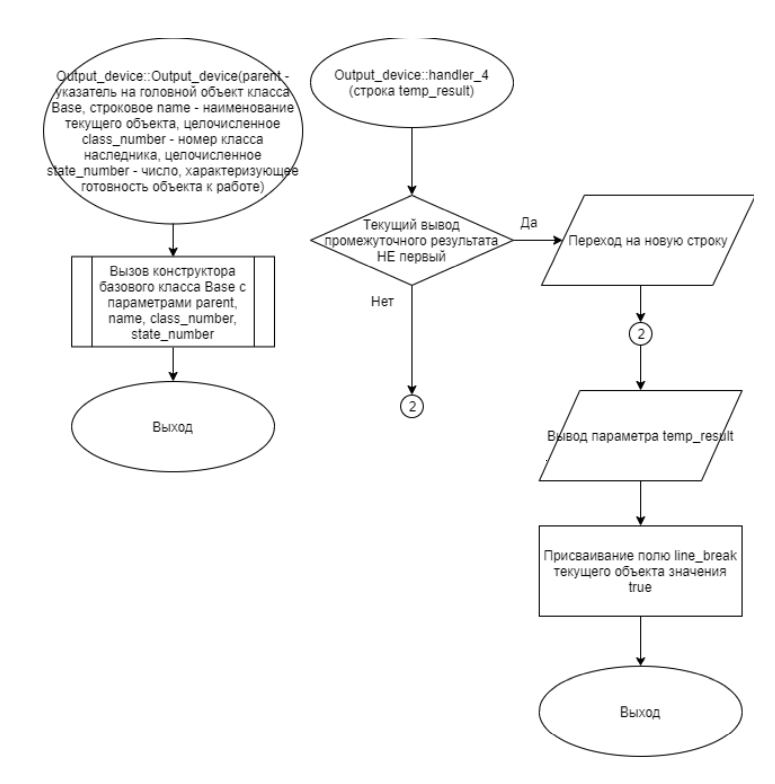


Рисунок 8 — Блок-схема алгоритма, часть 8

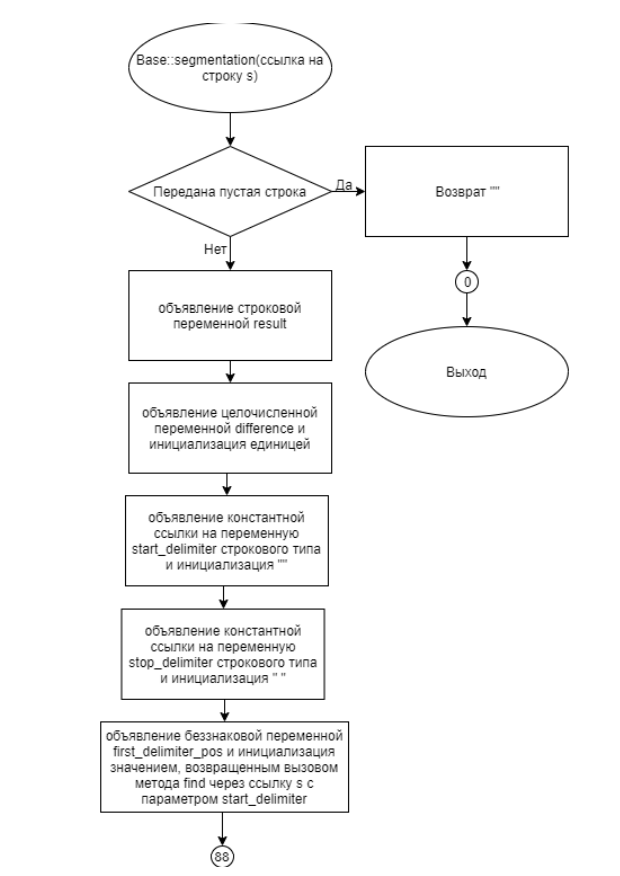


Рисунок 9 — Блок-схема алгоритма, часть 9

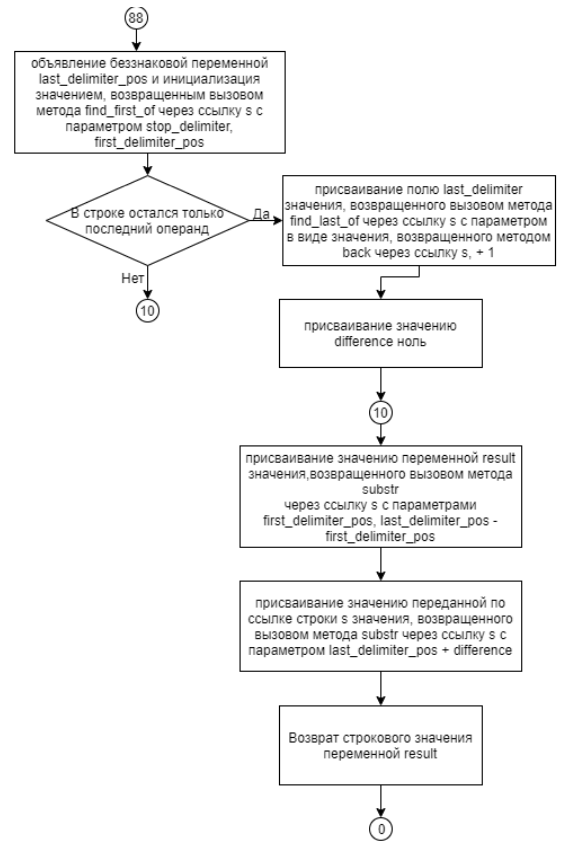


Рисунок 10 — Блок-схема алгоритма, часть 10

## Код программы

Программная реализация алгоритма решения задачи представлена в Приложении А.

## Тестирование

Результат тестирования программы представлен в таблице 16.

Таблица 16 - Результат тестирования программы.

|  |  |  |
| --- | --- | --- |
| Входные данные | Ожидаемые выходные данные | Фактические выходные данные |
| 5 + 6 - 1 \* 3 | 5 + 6 = 11  11 - 1 = 10   10 \* 3 = 30 | 5 + 6 = 11  11 - 1 = 10   10 \* 3 = 30 |
|  |  |  |
| 15 - 5 \* 9 % 15 - 404 + 505 | 15 - 5 = 10  10 \* 9 = 90 90 % 15 = 0 | 15 - 5 = 10  10 \* 9 = 90  90 % 15 = 0 |
|  | 0 - 404 = -404   -404 + 505 = 101 | 0 - 404 = -404   -404 + 505 = 101 |
|  |  |  |

# ЗАКЛЮЧЕНИЕ

В результате выполнения работы я приобрел навыки написания программ с помощью ООП подхода на примере языка программирования С++. В том числе на конкретных учебных задачах я научился разрабатывать базовый класс для объектов, определять общий функционал для используемых в рамках приложения объектов, разрабатывать операции добавления, удаления, изменения позиции объекта в рамках иерархического дерева, построению дерева иерархии объектов, переключению состояния объектов и определению их готовности к работе, выводить на печать дерево иерархии объектов, искать указатель на объект по координате на дереве иерархии объектов или по имени, при уникальности наименований объектов.

Также я получил навыки проектирования, написания, отладки и модификации программ в ООП стиле. Были изучены все стандартные конструкции и основные библиотеки языка С++, и усвоен материал, изложенный в учебных пособиях по данной теме. При возникновении трудностей обращался за консультацией к профессиональному сообществу на различных интернет-ресурсах, к своим семинаристам и лектору. Все полученые знания не раз применил на практике.

Объекто-ориентированный подход является наиболее распространенной методикой разработки на данный момент, и навык написания программм в этой парадигме является основополагающим для любого современного программиста. Данная курсовая работа помогла мне развить соответсвующие компетенции и приобрести необходимый опыт, который будет востребован при построении будущей карьеры и устройстве на работу.

# СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ (источников)

1. Васильев А.Н. Объектно-ориентированное программирование на С++. Издательство: Наука и Техника. Санкт-Петербург, 2016г. 543 стр.
2. Шилдт Г. С++: базовый курс. 3-е изд. Пер. с англ. — М.: Вильямс, 2017. — 624 с.
3. Методическое пособие для проведения практических заданий, контрольных и курсовых работ по дисциплине «Объектно-ориентированное программирование» [Электронный ресурс] // URL: https://mirea.aco-avrora.ru/student/files/methodichescoe\_posobie\_dlya\_laboratornyh\_rabot\_3.pdf (дата обращения 06.06.2021).
4. Приложение к методическому пособию студента по выполнению заданий в рамках курса «Объектно-ориентированное программирование» [Электронный ресурс] // URL: https://mirea.aco-avrora.ru/student/files/Prilozheniye\_k\_methodichke.pdf (дата обращения 06.06.2021).
5. Конспект лекций по дисциплине «Объектно-ориентированное программирование» [Электронный ресурс] // URL: https://mirea.aco-avrora.ru/student/files/Конспект\_лекций\_Объектно\_ориентированное\_программирование.docx (дата обращения 06.06.2021).
6. Антик М.И. Дискретная математика [Электронный ресурс]: Учебное пособие / Антик М.И., Казанцева Л.В. — М.: МИРЭА — Российский технологический университет, 2018 — 1 электрон. опт. диск (CD-ROM).

# ПРИЛОЖЕНИЕ А

## Файл Application.cpp

#include "Application.h"

using namespace std;

Application::Application(Base\* parent, int class\_number, int state\_number) : Base(parent,"",class\_number,state\_number) {

}

void Application::buildTree() {

setName("app");

Base\* obj\_op\_handler = new Operation\_handler(this, "obj\_op\_handler", 2, 1);

Base\* obj\_out = new Output\_device(this, "obj\_out", 3, 1);

this->set\_connection(SIGNAL\_D(&Application::signal\_1), obj\_op\_handler, HANDLER\_D(&Operation\_handler::handler\_1));

this->set\_connection(SIGNAL\_D(&Application::signal\_2), obj\_op\_handler, HANDLER\_D(&Operation\_handler::handler\_2));

this->set\_connection(SIGNAL\_D(&Application::signal\_3), obj\_op\_handler, HANDLER\_D(&Operation\_handler::handler\_3));

obj\_op\_handler->set\_connection(SIGNAL\_D(&Operation\_handler::signal\_4), obj\_out, HANDLER\_D(&Output\_device::handler\_4));

};

int Application::read\_data() {

if (this->getName() != "endtree") {

cout << endl << "Test result" << endl;

cout << "The object " << this->getName() << " is ready";

this->read\_base\_data();

}

return 0;

}

void Application::start\_emitting\_signals() {

cout << endl << "Emit signals";

while (true) {

string object\_signal\_name;

string text;

cin >> object\_signal\_name;

if (object\_signal\_name == "endsignals") {

break;

}

cin >> text;

Base\* signal = root->search\_by\_name(object\_signal\_name);

ptr\_s ptr\_this\_s = 0;

/\*switch (signal->class\_number) {

case 1: ptr\_this\_s = SIGNAL\_D(&Derived::send\_signal); break;

case 2: ptr\_this\_s = SIGNAL\_D(&Derived2::send\_signal); break;

case 3: ptr\_this\_s = SIGNAL\_D(&Derived3::send\_signal); break;

}\*/

signal->emit\_signal(ptr\_this\_s, text);

};

}

void Application::execute() {

string number1, number2, symbol;

string expression;

getline(cin, expression);

number1 = segmentation(expression); //определение первого операнда

emit\_signal(SIGNAL\_D(&Application::signal\_1), number1); //send\_signal-1

while (expression != "") {

symbol = segmentation(expression); //определение очередной операции

emit\_signal(SIGNAL\_D(&Application::signal\_2), symbol); //send\_signal-2

number2 = segmentation(expression); //определение очередного операнда

emit\_signal(SIGNAL\_D(&Application::signal\_3), number2);//send\_signal-3

}

};

## Файл Application.h

#ifndef APPLICATION\_H

#define APPLICATION\_H

#include "Base.h"

#include "Operation\_handler.h"

#include "Output\_device.h"

class Application : public Base {

Base\* root\_parent = 0;

Base\* temp\_child = 0;

public:

Application(Base\* parent = 0, int class\_number = 1, int state\_number = 1);

void buildTree();

void execute();

int read\_data();

void start\_emitting\_signals();

void signal\_1(string& nothing) {};

void signal\_2(string& nothing) {};

void signal\_3(string& nothing) {};

};

#endif

## Файл Base.cpp

#include "Base.h"

int Base::i;

Base\* Base::root;

Base::Base(Base\* parent, string name, int class\_number, int state\_number) {

i = 0;

setParent(parent);

setName(name);

setPath();

this->class\_number = class\_number;

this->state\_number = state\_number;

}

void Base::setName(string name) {

this->name = name;

}

string Base::getName() {

return this->name;

}

void Base::setPath() {

if (parent) {

this->path += parent->getPath();

}

if (name != "") {

this->path += "/" + this->name;

}

}

string Base::getPath() {

return this->path;

}

void Base::setParent(Base\* parent) {

this->parent = parent;

if (parent) {

parent->children.push\_back(this);

}

}

void Base::OutputTree() {

if (!i) { //условие при котором "шапка" выводится только один раз

cout << "Object tree";

cout << endl << name;

}

if (children.empty()) {

return;

}

children\_iterator = children.begin();

while (children\_iterator != children.end()) {

tabulation(1);

cout << endl << tabulation(0);

cout << (\*children\_iterator)->getName();

(\*children\_iterator)->OutputTree();

tabulation(-1);

children\_iterator++;

}

}

string Base::tabulation(int a) {

string this\_tabulation = "";

string tab = " ";

if (a == 1) {

++i;

}

if (a == -1) {

--i;

}

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

this\_tabulation.append(tab);

}

return this\_tabulation;

}

Base\* Base::getParent() {

return parent;

};

void Base::read\_base\_data() {

children\_iterator = children.begin();

while (children\_iterator != children.end()) {

cout << endl;

cout << "The object " << (\*children\_iterator)->getName();

if ((\*children\_iterator)->state\_number > 0) {

cout << " is ready";

}

else { cout << " is not ready"; }

(\*children\_iterator)->read\_base\_data();

children\_iterator++;

}

}

string Base::segmentation(string& s) {

if (s == "") {

return "";

}

string result;

int difference = 1; //to substract together with space

const string& start\_delimiter = ""; //there has been "/"

const string& stop\_delimiter = " "; //there has been "/"

unsigned first\_delimiter\_pos = s.find(start\_delimiter); //there has been + 1

unsigned last\_delimiter\_pos = s.find\_first\_of(stop\_delimiter, first\_delimiter\_pos);

if (last\_delimiter\_pos == pow(2, 32) - 1) {

last\_delimiter\_pos = s.find\_last\_of(s.back()) + 1;

difference = 0;

}

result = s.substr(first\_delimiter\_pos, last\_delimiter\_pos - first\_delimiter\_pos);

s = s.substr(last\_delimiter\_pos + difference);

return (result);

};

Base\* Base::search(string path) {

if (path.substr(0, 2) == "//") { //если адрес передан через //, то ищем по имени

string temp = path;

segmentation(temp); //убираем первый слэш

return search\_by\_name(segmentation(temp)); //убираем второй и вызываем поиск по имени

};

string local = segmentation(path); //достаем имя верхней директории

if (local == this->getName() && path == "") {

return this;

}

if (local == this->getName()) {

local = segmentation(path);

}

children\_iterator = children.begin();

while (children\_iterator != children.end()) {

if (local == (\*children\_iterator)->getName()) {

if (path == "") {

return \*children\_iterator;

}

else {

return (\*children\_iterator)->search(path);

}

}

children\_iterator++;

}

return NULL;

}

Base\* Base::search\_by\_name(string name) {

if (name == this->getName()) return this;

children\_iterator = children.begin();

while (children\_iterator != children.end()) {

if (name == (\*children\_iterator)->getName()) {

return (\*children\_iterator);

}

if ((\*children\_iterator)->search\_by\_name(name) != NULL) {

return (\*children\_iterator)->search\_by\_name(name);

}

children\_iterator++;

}

return NULL;

}

Base\* Base::search\_by\_path(string path) {

if (search(path)) {

string temp = "";

temp = search(path)->getPath() + " " + "Object name: " + search(path)->getName();

cout << endl << temp;

return search(path);

}

else {

string temp = "";

temp = path + " " + "Object not found";

cout << endl << temp;

return NULL;

}

}

void Base::set\_connection(ptr\_s p\_signal, Base\* p\_ob\_handler, ptr\_h p\_handler) {

o\_sh\* new\_struct = new o\_sh;

if (connections.size() > 0) {

connections\_iterator = connections.begin();

while (connections\_iterator != connections.end()) {

if ((\*connections\_iterator)->p\_signal == p\_signal && (\*connections\_iterator)->p\_Base == p\_ob\_handler && (\*connections\_iterator)->p\_handler == p\_handler) {

return;

}

connections\_iterator++;

}

}

new\_struct->p\_signal = p\_signal;

new\_struct->p\_Base = p\_ob\_handler;

new\_struct->p\_handler = p\_handler;

connections.push\_back(new\_struct);

};

void Base::delete\_connection(ptr\_s p\_signal, Base\* p\_ob\_handler, ptr\_h p\_handler) {

if (connections.empty()) return;

connections\_iterator = connections.begin();

while (connections\_iterator != connections.end()) {

if ((\*connections\_iterator)->p\_signal == p\_signal && ((\*connections\_iterator))->p\_Base == p\_ob\_handler

&& ((\*connections\_iterator))->p\_handler == p\_handler) {

connections.erase(connections\_iterator);

break;

}

connections\_iterator++;

}

};

void Base::emit\_signal(ptr\_s s\_ignal, string& s\_command) {

if (connections.empty()) return;

(this->\*s\_ignal) (s\_command); //посылаем строковый сигнал на преобразование

connections\_iterator = connections.begin();

while (connections\_iterator != connections.end()) {

if ((\*connections\_iterator)->p\_signal == s\_ignal) {

ptr\_h p\_handler = (\*connections\_iterator)->p\_handler; //метод обработчика

Base\* ob\_handler = (\*connections\_iterator)->p\_Base; //указатель на объект-обработчик

(ob\_handler->\*p\_handler)(s\_command); //вызов метода обработчика с параметром в виде уже преобразованного сигнала

}

connections\_iterator++;

}

};

## Файл Base.h

#ifndef BASE\_H

#define BASE\_H

#include <iostream>

#include <vector>

#include <cmath>

#include <string>

#include <tuple>

#define SIGNAL\_D( signal\_f ) ( ( void ( Base::\* ) ( string & ) ) ( ( signal\_f ) ) )

#define HANDLER\_D( handler\_f ) ( ( void ( Base::\* ) ( string ) ) ( ( handler\_f ) ) )

using namespace std;

class Base;

typedef void (Base::\* ptr\_s)(string&);

typedef void (Base::\* ptr\_h)(string);

class Base {

static int i;

string name;

Base\* parent;

int state\_number;

string path;

vector<Base\*> children;

vector<Base\*>::iterator children\_iterator;

struct o\_sh {

ptr\_s p\_signal;

Base\* p\_Base;

ptr\_h p\_handler;

};

vector<o\_sh\*> connections;

vector<o\_sh\*>::iterator connections\_iterator;

public:

static Base\* root;

int class\_number;

Base(Base\*, string, int class\_number, int state\_number);

string getPath();

string getName();

Base\* getParent();

Base\* search\_by\_name(string);

Base\* search\_by\_path(string);

void OutputTree();

void show\_all\_connections();

void set\_connection(ptr\_s, Base\* p\_ob\_handler, ptr\_h);

void delete\_connection(ptr\_s, Base\* p\_ob\_handler, ptr\_h);

void emit\_signal(ptr\_s, string& s\_command);

protected:

string tabulation(int);

void setName(string);

void setPath();

void setParent(Base\*);

Base\* search(string);

void read\_base\_data();

string segmentation(string&);

};

#endif

## Файл main.cpp

#include "Application.h"

int main() {

Application app;

app.buildTree();

app.execute();

return 0;

}

## Файл Operation\_handler.cpp

#include "Operation\_handler.h"

Operation\_handler::Operation\_handler(Base\* parent, string name, int class\_number, int state\_number) : Base(parent, name, class\_number, state\_number) {}

void Operation\_handler::handler\_1(string first\_operand) {

this->first\_operand = first\_operand;

temp\_result = first\_operand;

};

void Operation\_handler::handler\_2(string symbol) {

this->operation\_symbol = symbol[0];

};

void Operation\_handler::handler\_3(string second\_operand) {

this->second\_operand = second\_operand;

first\_operand = temp\_result;

switch (operation\_symbol) {

case '+': temp\_result = to\_string(stoi(temp\_result) + stoi(second\_operand)); break; //+

case '-': temp\_result = to\_string(stoi(temp\_result) - stoi(second\_operand)); break; //-

case '\*': temp\_result = to\_string(stoi(temp\_result) \* stoi(second\_operand)); break; //\*

case '%': temp\_result = to\_string(stoi(temp\_result) % stoi(second\_operand)); break; //%

}

string output = first\_operand + " " + operation\_symbol + " " + second\_operand + " = " + temp\_result;

emit\_signal(SIGNAL\_D(&Operation\_handler::signal\_4), output);

};

void Operation\_handler::signal\_4(string& output) {};

## Файл Operation\_handler.h

#ifndef OPERATION\_HANDLER\_H

#define OPERATION\_HANDLER\_H

#include "Base.h"

class Operation\_handler : public Base {

char operation\_symbol;

string first\_operand;

string second\_operand;

string temp\_result;

public:

Operation\_handler(Base\*, string, int, int);

void handler\_1(string);

void handler\_2(string);

void handler\_3(string);

void signal\_4(string&);

};

#endif

## Файл Output\_device.cpp

#include "Output\_device.h"

Output\_device::Output\_device(Base\* parent, string name, int class\_number, int state\_number) : Base(parent, name, class\_number, state\_number) {}

void Output\_device::handler\_4(string temp\_result) {

if (line\_break) {

cout << endl;

}

cout << temp\_result;

line\_break = true;

};

## Файл Output\_device.h

#ifndef OUTPUT\_DEVICE\_H

#define OUTPUT\_DEVICE\_H

#include "Base.h"

class Output\_device : public Base {

bool line\_break = false;

public:

Output\_device(Base\*, string, int, int);

void handler\_4(string);

};

#endif