Smart Calc v2.0

Лекция про калькулятор на C++

Сергей
“bernarda”

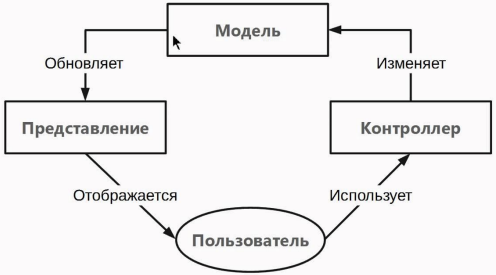
Содержание

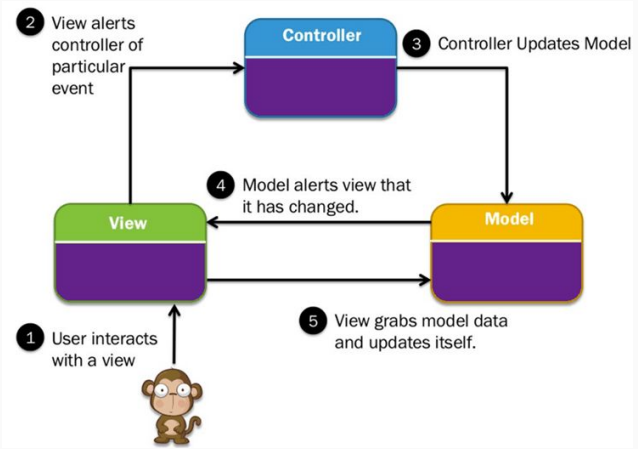
1. MVC - паттерн
2. MathCalculator Class
3. Token Class
4. Parsing
5. TokenMap
6. ConvertInfixToPostfix
7. PostfixNotationCalculation
8. GraphCalculation

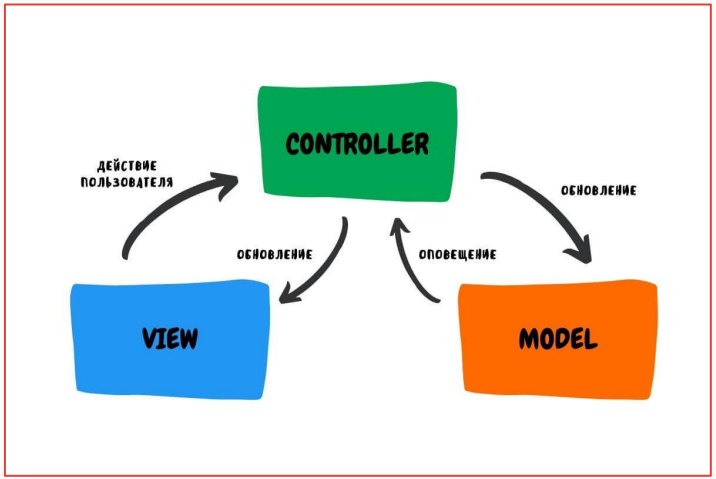
MVC - паттерн

Шаблон проектирования MVC (Model-View-Controller) разделяет приложение на три компонента:

- Модель (Model) представляет собой данные и бизнес-логику приложения.
- Представление (View) отвечает за отображение данных пользователю.
- Контроллер (Controller) обрабатывает пользовательский ввод и управляет взаимодействием между Моделью и Представлением.







MVC - паттерн

В C++, можно использовать MVC следующим образом:

- Модель (Model) может быть реализована в виде класса, который содержит данные и методы для их изменения.
- Представление (View) может быть реализовано с помощью класса, который отображает данные пользователю.
- Контроллер (Controller) может быть реализован в виде класса, который обрабатывает пользовательский ввод и вызывает соответствующие методы Модели и Представления.

```
void MainWindow::on_toolButton_equal_clicked() {
  try {
    controller_->CalculateValue(this);
  } catch (const std::exception &e) {
    QMessageBox::critical(this, "Warning", e.what());
  }
}
```

```
void Controller::CalculateValue(MainWindow *main_window) {
  model_math_->CalculateAnswer(main_window->GetInputString(),
                               main_window->GetInputStringX());
  main_window->SetAnswer(model_math_->GetAnswer());
}
```

Контроллер

```
void CalculateAnswer(const std::string& input_expression,
                     const std::string& input_x);
double GetAnswer() const;
```

Модель

```
std::string MainWindow::GetInputString() const {
  return ui->lineEdit_input->text().toStdString();
}

void MainWindow::SetAnswer(double x) {
  ui->lineEdit_output->setText(QString::number(x, 'g', 16));
}
```

Вид

Class MathCalculator

У класса есть несколько общедоступных функций-членов, которые выполняют вычисления и возвращают значения.

Функция `CalculateAnswer` принимает входное выражение и значение для `x`. Она вычисляет ответ на выражение и сохраняет его в члене-переменной `answer_`.

Функция `CalculateGraph` также принимает входное выражение, а также параметры для построения графика (количество точек, начальное и конечное значение x, минимальное и максимальное значение y). Она вычисляет график для выражения и сохраняет его в переменную-член `answer_graph_`.

Функция `GetAnswer` возвращает последний вычисленный ответ.

Функция `GetGraph` возвращает последний вычисленный график.

Остальные функции являются приватными и используются внутри класса для выполнения вычислений и преобразования выражения в постфиксную форму для последующего вычисления.

```
class MathCalculator {
public:
  using XYGraph = std::pair<std::vector<double>, std::vector<double>>;
  MathCalculator();
  ~MathCalculator() = default;

  void CalculateAnswer(const std::string& input_expression, const std::string& input_x);
  void CalculateGraph(const std::string& input_expression, int number_of_points, double x_start, double x_end, double y_min, double y_max);
  double GetAnswer() const;
  XYGraph GetGraph() const;

private:
  double answer_{NAN};
  XYGraph answer_graph_;

  std::string input_expression_;
  std::string input_x_;
  double x_value_{NAN};

  std::map<std::string, Token> token_map_;
  std::queue<Token> input_;
  std::vector<Token> stack_;
  std::queue<Token> output_;
  std::vector<double> result_;

  static constexpr bool kAdjacencyMatrix_[kNumTokenTypes][kNumTokenTypes] = {
     {0, 1, 0, 1, 0, 0, 1},  // kNumber
     {1, 0, 1, 0, 1, 1, 0},  // kBinaryOperator
     {1, 0, 1, 0, 1, 1, 0},  // kUnaryPrefixOperator
     {0, 1, 0, 1, 0, 0, 1},  // kUnaryPostfixOperator
     {0, 0, 0, 0, 0, 1, 0},  // kUnaryFunction
     {1, 0, 1, 0, 1, 1, 0},  // kOpenBracket
     {0, 1, 0, 1, 0, 0, 1},  // kCloseBracket
  };
  static constexpr bool kFirstToken_[kNumTokenTypes] = {1, 0, 1, 0, 1, 1, 0};
  static constexpr bool kLastToken_[kNumTokenTypes] = {1, 0, 0, 1, 0, 0, 1};

// другие private методы //
};
```

Token Class

Класс Token представляет токен для математических выражений. Он содержит информацию о имени, приоритете, ассоциативности, типе и функции, связанной с токеном. Токен может представлять число, бинарный оператор, префиксный оператор, постфиксный оператор, унарную функцию, открывающую скобку и закрывающую скобку.

Класс Token содержит методы для получения имени, приоритета, ассоциативности, типа и функции, связанной с токеном. Также он содержит методы для создания токенов для чисел и унарной операции отрицания.

Класс Token также используется для создания карты токенов.

```
class Token {
 public:
  Token() = default;
  Token(const std::string& name, Precedence precedence,
        Associativity associativity, Type type, function_variant function);
  ~Token() = default;

// геттеры

  void MakeNumber(std::string name, double value);
  void MakeUnaryNegation();

 private:
  std::string name_;                  // имя токена
  Precedence precedence_;             // приоритет операции
  Associativity associativity_;       // ассоциативность операции
  Type type_;                         // тип токена
  function_variant function_;         // функция токена
};
```

Enum

Три перечисления: `Type`, `Precedence`, и `Associativity`.

`Type` определяет типы токенов, которые могут встречаться в выражениях.

`Precedence` определяет приоритеты операций, которые используются для определения порядка выполнения операций в выражениях.

`Associativity` определяет ассоциативность операций, которая используется для разрешения неоднозначностей в выражениях с операциями одинакового приоритета.

```
enum Type {
  kNumber,
  kBinaryOperator,
  kUnaryPrefixOperator,
  kUnaryPostfixOperator,
  kUnaryFunction,
  kOpenBracket,
  kCloseBracket,
  kNumTokenTypes,
};
```

```
enum Precedence {
  kDefault,
  kLow,
  kMedium,
  kHigh,
  kUnaryOperator,
  kFunction,
};
```

```
enum Associativity {
  kNone,
  kLeft,
  kRight,
};
```

std::function
std::variant

Три псевдонима типа: unary_function, binary_function, и function_variant.

unary_function и binary_function являются псевдонимами типа std::function<double(double)> и std::function<double(double, double)> соответственно, то есть они представляют функции, принимающие один и два аргумента типа double.

function_variant является псевдонимом типа std::variant, который может хранить значения типов double, unary_function, binary_function, или nullptr_t. std::variant - это класс-обёртка, которая может хранить значения разных типов в единственном объекте.

```
using unary_function = std::function<double(double)>;
using binary_function = std::function<double(double, double)>;
using function_variant =
    std::variant<double, unary_function, binary_function, nullptr_t>;
```

Token Map

Этот код определяет функцию `CreateTokenMap`, которая принимает ссылку на словарь, отображающий строки на объекты типа `MyNamespace::Token`.

Затем она инициализирует и вставляет в словарь набор пар ключ-значение, где ключ - строка, а значение - объект `MyNamespace::Token`.

Каждый объект `MyNamespace::Token` представляет токен математического выражения, такой как скобки, операторы, функции и т. д.

+ Функции сразу записаны в токене
+ Чтобы добавить в калькулятор новую функцию - нужно добавить одну строку в map

```
#include <cmath>
#include <functional>
```

последний параметр токена

variant

double, unary_function, binary_function, nullptr_t

```
void MyNamespace::CreateTokenMap(
    std::map<std::string, MyNamespace::Token>& token_map) {
  using std::initializer_list;
  using std::pair;
  using std::string;
  using namespace MyNamespace;
  initializer_list<pair<const string, Token>> list = {
      {" ", Token("space", kDefault, kNone, kNumber, nullptr)},
      {"x", Token("x", kDefault, kNone, kNumber, nullptr)},
      {"(", Token("(", kDefault, kNone, kOpenBracket, nullptr)},
      {")", Token(")", kDefault, kNone, kCloseBracket, nullptr)},
      {"+", Token("+", kLow, kLeft, kBinaryOperator, std::plus<double>())},
      {"-", Token("-", kLow, kLeft, kBinaryOperator, std::minus<double>())},
      {"*",
       Token("*", kMedium, kLeft, kBinaryOperator, std::multiplies<double>())},
      {"/",
       Token("/", kMedium, kLeft, kBinaryOperator, std::divides<double>())},
      {"^", Token("^", kHigh, kRight, kBinaryOperator, powl)},
      {"mod", Token("mod", kMedium, kLeft, kBinaryOperator, fmodl)},
      {"cos", Token("cos", kFunction, kRight, kUnaryFunction, cosl)},
      {"sin", Token("sin", kFunction, kRight, kUnaryFunction, sinl)},
      {"tan", Token("tan", kFunction, kRight, kUnaryFunction, tanl)},
      {"acos", Token("acos", kFunction, kRight, kUnaryFunction, acosl)},
      {"asin", Token("asin", kFunction, kRight, kUnaryFunction, asinl)},
      {"atan", Token("atan", kFunction, kRight, kUnaryFunction, atanl)},
      {"sqrt", Token("sqrt", kFunction, kRight, kUnaryFunction, sqrtl)},
      {"ln", Token("ln", kFunction, kRight, kUnaryFunction, logl)},
      {"log", Token("log", kFunction, kRight, kUnaryFunction, log10l)},
      {"cbrt", Token("cbrt", kFunction, kRight, kUnaryFunction, cbrtl)},
      {"exp", Token("exp", kFunction, kRight, kUnaryFunction, expl)},
      {"abs", Token("abs", kFunction, kRight, kUnaryFunction, fabsl)},
      {"round", Token("round", kFunction, kRight, kUnaryFunction, roundl)},
      {"e", Token("e", kDefault, kNone, kNumber, M_E)},
      {"pi", Token("pi", kDefault, kNone, kNumber, M_PI)},
      {"inf", Token("inf", kDefault, kNone, kNumber, INFINITY)},
      {"!",
       Token("!", kUnaryOperator, kLeft, kUnaryPostfixOperator, factorial)},
  };
  token_map.insert(list);
}
```

Token Class

Методы для создания
токена *числа*
токена *унарного минуса*

```
void MyNamespace::Token::MakeNumber(std::string name, double value) {
  Token result(name, kDefault, kNone, kNumber, value);
  *this = result;
}

void MyNamespace::Token::MakeUnaryNegation() {
  Token result("negate", kUnaryOperator, kRight, kUnaryPrefixOperator,
               std::negate<double>());
  *this = result;
}
```

Предварительная обработка строки

`tolower` – переводит символ в нижний регистр

`transform` – применяет функцию к каждому символу в троке

```
std::string MyNamespace::MathCalculator::ConvertToLowercase(std::string str) {
  std::transform(str.begin(), str.end(), str.begin(), ::tolower);
  return str;
}
```

Парсер строки

Идем по строке и считываем символы

1) `isalpha` – является ли символ буквой -> читаем слово
2) `isdigit` – является ли символ цифрой -> читаем число
3) иначе это одиночный символ

```
std::string MyNamespace::MathCalculator::ReadWord(std::string& input,
                                                   size_t& start_index) const {
  std::regex word_regex("([a-z]+)");
  std::sregex_iterator regex_iterator = std::sregex_iterator(
      input.begin() + start_index, input.end(), word_regex);
  std::smatch match = *regex_iterator;
  start_index += match.length();
  return match.str();
}
```

```
std::regex double_regex("\\d+([.]\\d+)?(e([-+])?\\d+)?");
```

В C++ есть несколько способов преобразовать строку в число. Ниже приведены некоторые из них.

1. С помощью функции `std::stoi` (или `std::stof`, `std::stod` для других типов):
2. С помощью потоков ввода-вывода `std::stringstream`
3. С помощью функции `std::from_chars` (с C++17):

Пушим токены в очередь

Функция принимает строковый аргумент `token`.

Внутри функции выполняется поиск `token` в `token_map_`.

Если `token` не найден, то выбрасывается исключение `std::logic_error` с сообщением "Incorrect symbol: " и содержимым `token`.

Если `token` найден, то его значение добавляется в `input_`.

```
void MyNamespace::MathCalculator::TryToPushToken(std::string token) {
  auto token_map_it = token_map_.find(token);
  if (token_map_it == token_map_.end()) {
    throw std::logic_error("Incorrect symbol: " + token);
  }
  input_.push(token_map_it->second);
}
```

можно делать и наоборот:
искать ключи map во входной строке

Обработка очереди токенов

1. Удалить пробелы
2. Поменять бинарные - и + на унарные - и +
3. Сделать вставки токенов умножения (по заданию необязательно)
4. Проверить последовательность токенов

Пробелы удаляем только сейчас
Удалять их до разделения строки на токены - ошибка

Бинарные - и + идут после токенов числа и закрывающейся скобки и *посфиксного оператора**
Все остальные - и + необходимо перевести в унарные

Здесь же можно делать вставки токена умножения, чтобы была поддержка ввода:
2sin(5x)cos(4(1+x))
2*sin(5*x)*cos(4*(1+x))

** - не требуется по заданию*

Матрица смежности

Статический двумерный массив `kAdjacencyMatrix_`, представляет матрицу смежности для токенов, используемых в математическом выражении.

Значение `true` в ячейке матрицы означает, что соответствующие токены могут находиться рядом друг с другом в выражении, а значение `false` означает, что они не могут.

Например, ячейка `kAdjacencyMatrix_[0][1]` равна `1`, что означает, что число (токен `kNumber`) может следовать за бинарным оператором (токен `kBinaryOperator`) в выражении.

Для первого и последнего токена проверки делаем отдельно.

```
std::queue<Token> input_;
std::queue<Token> output_;
```

приватные поля класса MathCalculator

```
enum Type {
  kNumber,
  kBinaryOperator,
  kUnaryPrefixOperator,
  kUnaryPostfixOperator,
  kUnaryFunction,
  kOpenBracket,
  kCloseBracket,
  kNumTokenTypes,
};
```

```
static constexpr bool kAdjacencyMatrix_[kNumTokenTypes][kNumTokenTypes] = {
    {0, 1, 0, 1, 0, 0, 1},  // kNumber
    {1, 0, 1, 0, 1, 1, 0},  // kBinaryOperator
    {1, 0, 1, 0, 1, 1, 0},  // kUnaryPrefixOperator
    {0, 1, 0, 1, 0, 0, 1},  // kUnaryPostfixOperator
    {0, 0, 0, 0, 0, 1, 0},  // kUnaryFunction
    {1, 0, 1, 0, 1, 1, 0},  // kOpenBracket
    {0, 1, 0, 1, 0, 0, 1},  // kCloseBracket
};
```

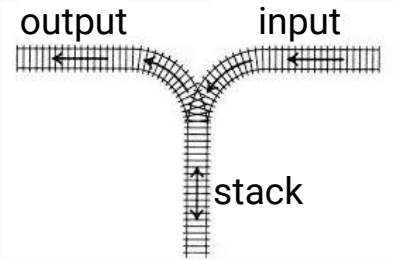
```
for (; !output_.empty() && !input_.empty(); MoveTokenFromInputToOutput()) {
  if (!kAdjacencyMatrix_[output_.back().GetType()]
                        [input_.front().GetType()]) {
    throw std::logic_error("Wrong sequence: " + output_.back().GetName() +
                           " " + input_.front().GetName());
  }
}
```

Постфиксная нотация

- “Обратная польская нотация”
- Алгоритм сортировочной станции (Википедия) - Алгоритм Дейкстры
- Самая длинная функция в коде (~50 строк)
- Активно используем разные свойства токенов, чтобы код выглядел как текст алгоритма

```
void MyNamespace::MathCalculator::ConvertInfixToPostfix() {
  using namespace MyNamespace;
  while (!input_.empty()) {
    switch (input_.front().GetType()) {
      case Type::kNumber:
      case Type::kUnaryPostfixOperator:
        MoveTokenFromInputToOutput();
        break;
      case Type::kUnaryFunction:
      case Type::kUnaryPrefixOperator:
      case Type::kOpenBracket:
        MoveTokenFromInputToStack();
        break;
      case Type::kBinaryOperator:
```

Дальше сами =)



Вычисление с помощью std::visit

`std::visit` - это шаблонная функция в C++, которая позволяет вызывать различные функции в зависимости от типа объекта варианта (`std::variant`).

В данном контексте `std::visit` используется для обработки токенов, хранящихся в очереди `input_` в методе `PostfixNotationCalculation`.

Функция `std::visit` принимает в себя вариант (variant) и набор лямбда-выражений, которые будут вызываться в зависимости от типа варианта. Лямбда-выражения принимают в себя аргументы, соответствующие типу варианта, и выполняют определенные действия.

В данном методе лямбда-выражения выполняют различные действия в зависимости от типа элемента на вершине очереди `input_`.

`std::visit` удобно использовать вместо многочисленных условных операторов `if-else` или `switch-case` для обработки объектов разных типов.

```
using unary_function = std::function<double(double)>;
using binary_function = std::function<double(double, double)>;
using function_variant =
    std::variant<double, unary_function, binary_function, nullptr_t>;
```

```
double MyNamespace::MathCalculator::PostfixNotationCalculation(double x_value) {
  using namespace MyNamespace;
  input_ = output_;
  while (!input_.empty()) {
    std::visit(
        overloaded{[&](double function) { PushToResult(function); },
                   [&](unary_function function) {
                     PushToResult(function(PopFromResult()));
                   },
                   [&](binary_function function) {
                     double right_argument = PopFromResult();
                     double left_argument = PopFromResult();
                     PushToResult(function(left_argument, right_argument));
                   },
                   [&](auto) { PushToResult(x_value); }},
        input_.front().GetFunction());
  }
  return PopFromResult();
}
```

Вычисление с помощью std::visit

Данный код определяет шаблонный класс `overloaded`, который используется для переопределения лямбда-выражений в `std::visit`.

Класс `overloaded` наследуется от переданных ему шаблонных параметров `Ts` и перегружает оператор вызова `operator()`. Это позволяет создать объект, который может быть передан в `std::visit` в качестве набора лямбда-выражений.

Далее определен шаблонный конструктор, который принимает шаблонные параметры `Ts`. Он создает объект класса `overloaded` с переданными параметрами `Ts`.

Такой подход позволяет удобно переопределять лямбда-выражения и использовать их в `std::visit`. Вместо того, чтобы передавать каждое лямбда-выражение отдельно, можно передать объект класса `overloaded`, который уже содержит перегруженные операторы вызова для каждого из переданных лямбда-выражений.

копируем из из документации std::visit

```
/// @brief template class for redefining lambda expressions in std::visit
/// @tparam ...Ts - accepted type of lambda expression
template <class... Ts>
struct overloaded : Ts... {
  using Ts::operator()...;
};

/// @brief method of redefining the list of arguments of the overloaded method
/// into classes
/// @tparam ...Ts - accepted type of lambda expression
template <class... Ts>
overloaded(Ts...) -> overloaded<Ts...>;
};  // namespace MyNamespace
```

График

Сначала один раз все методы кроме вычисления

CalculateXY - заполнение векторов X и Y значениями

В цикле только метод `PostfixNotationCalculation`

```
using XYGraph = std::pair<std::vector<double>, std::vector<double>>;

XYGraph MyNamespace::MathCalculator::GetGraph() const {
 return answer_graph_;
}

void MyNamespace::MathCalculator::CalculateXY(int number_of_points,
        double x_start, double x_end, double y_min, double y_max) {

  std::vector<double> x_values, y_values;
  double step = (x_end - x_start) / (number_of_points - 1);
  for (int i = 0; i < number_of_points; ++i) {
    x_values.push_back(x_start + step * i);
    y_values.push_back(PostfixNotationCalculation(x_values.back()));
  }
  answer_graph_ = std::make_pair(x_values, y_values);
}


 // чтобы сделать разрыв графика
 // можно пушить в y_values std::numeric_limits<double>::quiet_NaN()
```

"Разработка проекта не может быть закончена, она может быть остановлена"

– Семён “theiaesp”

Спасибо!

Контактная информация:

Сергей “bernarda”

Новосибирск - 15 волна