**JSON ПАРСЕР**

*от Апостол Янков*

1. **Въведение**

JSON (JavaScript Object Notation) е свободно-достъпен стандартен файлов формат за съхранение и обмен на данни, който е удобен за четене и работа, поради което е и добра алтернатива на XML формата. JSON файловете използват разширението “.json”, а тяхната употреба влиза в много съвременни езици за програмиране. Най-често JSON файловете се използват за комуникация между клиент и сървър чрез заявки.

JSON данните се записват като двойки ключ/стойност, като те са разделени, чрез двуеточие “:“ (ключът е отляво, стойността - отдясно), а стойностите могат да бъдат следните типове:

* число – short, int, long, double, etc.
* низ – поредица от Unicode символи (string)
* булева стойност – вярно/невярно
* масив – списък от стойности
* обект – колекция от двойки ключ/стойност

При работа с JSON файлове се срещат термините сериализация и десериализация. Сериализацията представлява превръщане на обект (колекция от двойки ключ/стойност) към низ (string), и се използва, когато искаме да подадем данни към някой JSON файл за дадена операция. Десериализацията е обратният процес на превръщане на текст в обект – използва се, когато искаме да прочетем даден JSON файл и да запазим данните от него в наша структура в кода.

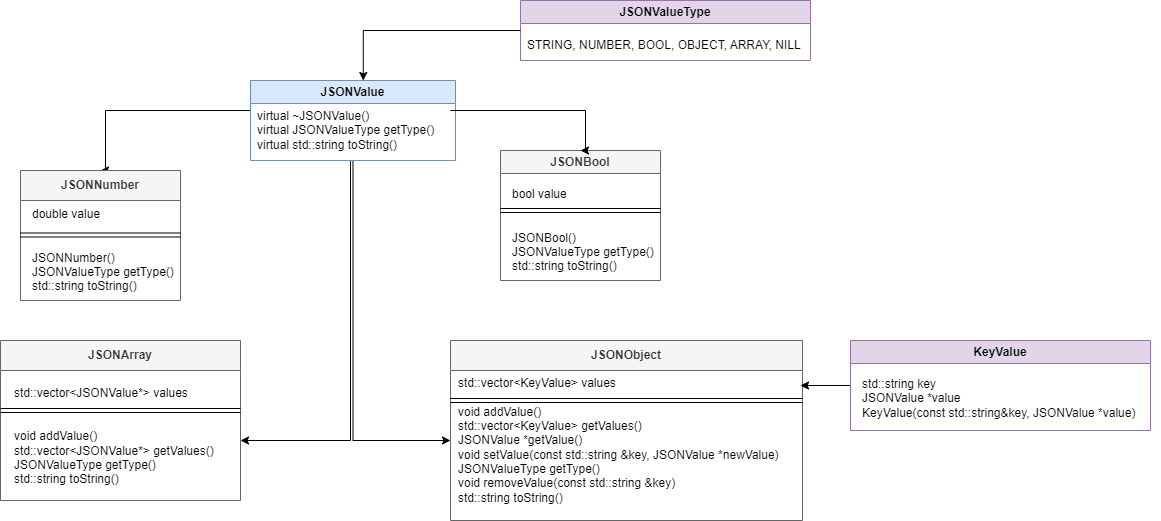
1. **Преглед на предметната дейност**

Този проект реализира JSON парсер на C++ без използване на външни библиотеки. Парсерът може да валидира, манипулира и изпълнява различни операции върху JSON данни. Ключовите компоненти на проекта са класовете Lexer, Token, Parser, JSONValue и Engine.

Главният проблем, който трябва да решим, е как да прочитаме и обработваме JSON файловете, така че да можем да ги променяме като обект в кода, след което да ги презапишем отново във файл. Трябва да можем рекурсивно да итерираме нивата на JSON файловете, за да четем, записваме, променяме и трием данни.

Част от този проблем е откриването на подходящ начин за пазене на тези данни в обект. Тоест какви структури от данни да използваме, че да го направим възможно. В този проект това се решава чрез основен клас JSONValue, който може да бъде всеки един тип, който може да бъде срещнат в един JSON файл, представен чрез отделно поле за всеки тип. Така всеки обект JSONValue има попълнено съответно едно от тези полета, спрямо това какъв е типът му.

**Компоненти**

****

*Диаграма 1.*

**1) Lexer**

Класът Lexer е отговорен за лексикалния анализ на JSON входа. Той токенизира входния низ в смислени токени, които парсерът Parser може да обработи.

**Публични методи:**

* Lexer(const std::string &input): Конструира обект Lexer с дадения вход.
* size\_t getLine(): Връща текущия номер на реда.
* size\_t getColumn(): Връща текущия номер на колоната.
* Token nextToken(): Взема следващия токен от входа.
* void resetPos(): Нулира позицията до началото на входа.

**Частни методи:**

* void advance(): Премества текущата позиция с един символ напред.
* void skipWhitespace(): Пропуска празни символи във входа.
* Token parseString(): Парсира низов токен.
* Token parseNumber(): Парсира числов токен.
* Token parseKeyword(): Парсира ключова дума (true, false, null).

**2) Token**

Класът Token представлява токен в JSON с тип и стойност.

**Енумерация TokenType:**

* LEFT\_BRACE, RIGHT\_BRACE, LEFT\_BRACKET, RIGHT\_BRACKET, COMMA, COLON, STRING, NUMBER, TRUE, FALSE, NULL\_TYPE, END

**Структура Token:**

* TokenType type: Тип на токена.
* std::string value: Стойност на токена, ако е приложимо.

**3) JSONValue**

Класът JSONValue представлява стойност в JSON, която може да бъде обект, масив, низ, число, булева стойност или null. Той е виртуален и бива наследен от всички останали подтипове (JSONArray, JSONObject, JSONNumber, JSONBool, JSONNull).

**Публични методи:**

* ~JSONValue(): виртуален деструктор.
* virtual std::string toString() const: виртуален метод, който преобразува всеки наследник в символен низ.
* virtual JSONValueType getType() const: виртуален метод, който връща типа на класа наследник.

**JSONNumber**

Притежава поле от тип double, съдържащо стойността на числото. Освен това има параметризиран конструктор за инициализация на стойността. При toString() се преценява дали числото е цяло или с плаваща запетая и прави съответната трансформация.

**JSONBool**

Притежава поле от тип bool, съдържащо стойността (true/false). Освен това има параметризиран конструктор за инициализация на стойността.

**JSONNull**

**JSONArray**

Притежава поле вектор от JSONValue \*, съдържащо стойностите в JSON масива. Допълнително има метод за добавяне на стойност и още един за връщане на всички стойности.

**JSONObject**

Притежава поле вектор от KeyValue, съдържащо двойки ключ-стойност, представляващи JSON обекта. Има методи:

* за връщане на всички стойности
* за добавяне на нова стойност
* за добавяне и замяна на стойност
* за връщане на стойност по подаден ключ от тип string
* за премахване на стойност по подаден ключ от тип string

**4) Parser**

Класът Parser е отговорен за парсиране, манипулиране и валидиране на JSON данни.

**Публични методи:**

* Parser(const std::string &input): Конструктор с даден входен низ.
* JSONValue parse(): Парсира входния JSON и връща кореновата JSON стойност.
* bool validate(): Валидира JSON структурата.
* std::vector<JSONValue \*> searchKey(const std::string &key) const: Търси ключ в JSON структурата и връща вектор от съответстващи JSON стойности.
* bool contains(const std::string &value) const: Проверява дали стойност се съдържа в JSON структурата.
* bool set(const std::string &path, const std::string &newValue): Задава нова стойност на даден път в JSON структурата.
* bool create(const std::string &path, const std::string &newValue): Създава нов елемент на даден път в JSON структурата.
* bool deleteElement(const std::string &path): Изтрива елемент на даден път в JSON структурата.
* bool move(const std::string &from, const std::string &to): Премества елементи от един път на друг в JSON структурата.
* bool save(const std::string &currPath, const std::string &filePath = "", const std::string &path = ""): Запазва JSON структурата във файл.
* bool saveas(const std::string &currPath, const std::string &newFilePath, const std::string &path = ""): Запазва JSON структурата в указан файл.
* void writeToFile(const std::string &filePath): Записва JSON структурата във файл.

**Частни методи:**

* void writeJSON(std::ostream &outFile, JSONValue \*value, int indent = 0) const: Записва JSON стойност в изходен поток с интервали.
* JSONValue \*findValueByPath(const std::string &path): Намира JSON стойност по даден път.

Методи за десериализация. Основният метод е parseValue(), който спрямо типа на текущия токен извиква някой от другите методи за десериализация на съответния тип данни.

* JSONValue\* parseValue(): Парсира JSON стойност.
* JSONObject\* parseObject(): Парсира JSON обект.
* JSONArray\* parseArray(): Парсира JSON масив.
* bool containsHelper(const JSONValue \*jsonValue, const std::string &value) const: Помощна функция за проверка дали стойност се съдържа в JSON стойност.
* std::vector<std::string> splitPath(const std::string &path) const: Разделя низов път на отделни ключове.

**Searcher**

Осигурява логиката за търсене на стойности по ключ в JSONObject. Чрез два статични частни метода подсигурява търсенето в обект и масив. Главният метод searchByKey се използва от Parser класа в метода searchKey.

**Validator**

Методи за валидация на JSON файла. Главният метод е validateValue(), който спрямо типа на текущия токен извиква някой от другите методи за валидация на съответния тип данни.

* void validateValue(): Валидира JSON стойност.
* void validateObject(): Валидира JSON обект.
* void validateArray(): Валидира JSON масив.
* void validateString(): Валидира JSON низ.
* void validateNumber(): Валидира JSON число.

**Printer**

Съдържа методи за принтиране на всички типове данни, като главният метод е:

* static void print(const JSONValue \*jsonValue, int indentLevel = 0): Отпечатва JSON стойност.

**5) Engine**

Класът Engine е отговорен за обработка на входа от потребителя и изпълнение на команди за манипулиране на JSON данни с помощта на парсера.

**Публични методи:**

* Engine(): Конструктор на Engine.
* void prompt(): Подканва потребителя за команди и ги изпълнява. Ако няма зареден файл, първо подканва потребителя да въведе пътя до JSON файла, с който ще работи.

**Частни методи:**

* void executeCommand(const std::string &command): Изпълнява дадена команда.
* void openFile(const std::string &filePath): Отваря указания файл и зарежда съдържанието му в парсера. Ако файлът не съществува, създава нов файл с празно съдържание.

**Основни части от кода**

Класът JSONValue и структурата KeyValue – един обект представлява вектор от двойки ключ/стойност (KeyValue). Масивът е представен като вектор от JSONValue указатели.

*enum class JSONValueType*

*{*

*STRING,*

*NUMBER,*

*BOOL,*

*OBJECT,*

*ARRAY,*

*NILL*

*};*

*class JSONValue*

*{*

*public:*

*virtual ~JSONValue() = default;*

*virtual JSONValueType getType() const = 0;*

*virtual std::string toString() const = 0;*

*};*

*Извадка 1.*

Класът Parser съдържа основната логика за сериализация/десериализация, както и за командите необходими за удовлетворяването на условието на проекта.

*writeJSON – попълва даден поток с данните в подходящ формат от JSONValue обект*

void Parser::writeJSON(std::ostream &out, const JSONValue &value, int indent = 0) const

{

    std::string indentStr(indent, ' ');

    switch (value.type)

    {

    case JSONValueType::OBJECT:

        out << "{\n";

        for (size\_t i = 0; i < value.objectValue.size(); ++i)

        {

            out << indentStr << "  \"" << value.objectValue[i].key << "\": ";

            writeJSON(out, \*value.objectValue[i].value, indent + 2);

            if (i < value.objectValue.size() - 1)

                out << ",";

            out << "\n";

        }

        out << indentStr << "}";

        break;

    case JSONValueType::ARRAY:

        out << "[\n";

        for (size\_t i = 0; i < value.arrayValue.size(); ++i)

        {

            out << indentStr << "  ";

            writeJSON(out, \*value.arrayValue[i], indent + 2);

            if (i < value.arrayValue.size() - 1)

                out << ",";

            out << "\n";

        }

        out << indentStr << "]";

        break;

    case JSONValueType::STRING:

        out << "\"" << value.stringValue << "\"";

        break;

    case JSONValueType::NUMBER:

        out << value.numberValue;

        break;

    case JSONValueType::BOOL:

        out << (value.boolValue ? "true" : "false");

        break;

    case JSONValueType::NIL:

        out << "null";

        break;

    default:

        throw std::runtime\_error("Unknown JSONType encountered in writeJSON.");

    }

}

*Извадка 2.*

Основният метод parseValue() се грижи да преобразува стрингов вход в JSONValue обект (десериализация). Извадки са показани от кода на parseValue() и parseObject(). Аналогично за останалите типове данни съществува десериализиращ метод, ползван в parseValue(). Забелязва се, че за обхождане на входния стринг се използва обект Lexer, от който също така се извлича информация за текущ ред и колона – полезно за дебъг от потребителя.

JSONValue \*Parser::parseValue()

{

    Token token = lexer.nextToken();

    switch (token.type)

    {

        case TokenType::LEFT\_BRACE:

            return parseObject();

        case TokenType::LEFT\_BRACKET:

            return parseArray();

        case TokenType::STRING:

            return new JSONString(token.value);

        case TokenType::NUMBER:

            return new JSONNumber(std::stod(token.value));

        case TokenType::TRUE:

            return new JSONBool(true);

        case TokenType::FALSE:

            return new JSONBool(false);

        case TokenType::NULL\_TYPE:

            return new JSONNull();

        default:

            throw std::runtime\_error("Unexpected token: " + token.value);

    }

}

*Извадка 3.*

JSONValue Parser::parseObject()

{

    JSONValue objectValue;

    objectValue.type = JSONValueType::OBJECT;

    currentToken = lexer.nextToken();

    if (currentToken.type != TokenType::RIGHT\_BRACE)

    {

        while (true)

        {

            if (currentToken.type != TokenType::STRING)

            {

                throw std::runtime\_error("Expected string key");

            }

            std::string key = currentToken.value;

            currentToken = lexer.nextToken();

            if (currentToken.type != TokenType::COLON)

            {

                throw std::runtime\_error("Expected ':'");

            }

            currentToken = lexer.nextToken();

            objectValue.objectValue.push\_back(KeyValue(key, new JSONValue(parseValue())));

            if (currentToken.type == TokenType::COMMA)

            {

                currentToken = lexer.nextToken();

            }

            else

            {

                break;

            }

        }

        if (currentToken.type != TokenType::RIGHT\_BRACE)

            throw std::runtime\_error("Expected '}' at line " + std::to\_string(lexer.getLine()) + ", column: " + std::to\_string(lexer.getColumn()));

    }

    currentToken = lexer.nextToken();

    return objectValue;

}

*Извадка 4.*

Ключова част от Lexer класа е именно методът nextToken(), чрез който се обхожда даден стринг.

Token Lexer::nextToken()

{

    skipWhitespace();

    if (pos >= input.length())

    {

        return {TokenType::END, ""};

    }

    char curr = input[pos];

    switch (curr)

    {

    case '{':

        pos++;

        return {TokenType::LEFT\_BRACE, "{"};

    case '}':

        pos++;

        return {TokenType::RIGHT\_BRACE, "}"};

    case '[':

        pos++;

        return {TokenType::LEFT\_BRACKET, "["};

    case ']':

        pos++;

        return {TokenType::RIGHT\_BRACKET, "]"};

    case ',':

        pos++;

        return {TokenType::COMMA, ","};

    case ':':

        pos++;

        return {TokenType::COLON, ":"};

    case '"':

        return parseString();

    case 't':

    case 'f':

    case 'n':

        return parseKeyword();

    default:

        if (isdigit(curr) || curr == '-')

        {

            return parseNumber();

        }

        throw std::runtime\_error("Unexpected character at line " + std::to\_string(line) + ", column " + std::to\_string(column));

    }

}

*Извадка 5.*

Следните методи се грижат за обработването на низове, числа и други ключови думи като true, false и null в *nextToken()* от Извадка 5.

Token Lexer::parseString()

{

    size\_t start = pos + 1;

    advance();

    while (pos < input.size() && input[pos] != ‘”’)

    {

        advance();

    }

    if (pos >= input.size())

    {

        throw std::runtime\_error(“Unterminated string at line “ + std::to\_string(line) + “, column “ + std::to\_string(column));

    }

    size\_t end = pos;

    advance();

    return {TokenType::STRING, input.substr(start, end – start)};

}

Token Lexer::parseNumber()

{

    size\_t start = pos;

    while (pos < input.size() && (isdigit(input[pos]) || input[pos] == ‘.’ || input[pos] == ‘-‘ || input[pos] == ‘+’))

    {

        advance();

    }

    return {TokenType::NUMBER, input.substr(start, pos – start)};

}

Token Lexer::parseKeyword()

{

    size\_t start = pos;

    while (pos < input.size() && isalpha(input[pos]))

    {

        advance();

    }

    std::string keyword = input.substr(start, pos – start);

    if (keyword == “true”)

        return {TokenType::TRUE, “true”};

    if (keyword == “false”)

        return {TokenType::FALSE, “false”};

    if (keyword == “null”)

        return {TokenType::NULL\_TYPE, “null”};

    throw std::runtime\_error(“Invalid keyword ‘” + keyword + “’ at line “ + std::to\_string(line) + “, column “ + std::to\_string(column));

}

*Извадка 6.*

1. **Заключение**

Проектът JSON парсер в C++ демонстрира как може да се реализира мощен инструмент за манипулиране на JSON данни без използването на външни библиотеки. Ключовите компоненти, включващи класовете Lexer, Token, Parser, JSONValue и Engine, работят съвместно, за да осигурят ефективно парсване, валидиране и модификация на JSON документи.

В бъдеще могат да бъдат направени подобрения по потребителския интерфейс, както и да бъдат добавени методи по-подходящи за разработка на код.

**Използвана литература:**

<https://www.json.org/json-bg.html>

<https://cplusplus.com/reference>

<https://stackoverflow.com/>