JSON Parse

от Цветомир Стайков

1MI0800469 2 група 1 курс

1. Увод

1.1 Описание и идея на проекта

Проектът представлява програма в стил конзолен интерфейс *(Command Line Interface)*, в която потребителят може да зарежда *JSON (JavaScript Object Notation)* файлове. Тя дава възможност за манипулиране, извеждане и проверка на данните от файловете. Програмата е написана в обектно-ориентиран стил и по начин, който позволява части от нея да бъдат имплементирани в други проекти.

1.2 Цел и задача на разработка

Проектът трябва да дава възможност за манипулиране, проверка и извеждане на *JSON* файлове. Той трябва да има команди за проверка на валидността на заредения файл, които проверяват дали спазва *JSON* формата, да извежда информацията от файла в четим формат *(„pretty print“)*, да модифицира информацията и да я запазва както в текущия файл, така и в нов. Програмата трябва да бъде написана на *C++* в обектно-ориентиран формат и да бъде тествана чрез библиотека за тестване."

1.3 Структура на документация

- 1. Увод
    1.1. Описание и идея на проекта

    1.2. Цел и задачи на разработката

    1.3. Структура на документацията

- 2. Преглед на предметната област

    2.1. Основни дефиниции, концепции и алгоритми, които ще бъдат използвани

    2.2. Дефиниране на проблеми и сложност на поставената задача

    2.3. Подходи, методи (евентуално модели и стандарти) за решаване на поставените проблемите

    2.4 Потребителски (функционални) изисквания (права, роли, статуси, диаграми, ...) и качествени (нефункционални) изисквания (скалируемост, поддръжка, ...)

- 3. Проектиране

    3.1. Обща архитектура – ООП дизайн

3.2. Диаграми (на структура и поведение - по обекти, слоеве с най-важните извадки от кода)

- 4. Реализация, тестване

4.1. Реализация на класове (включва важни моменти от реализацията на класовете и малки фрагменти от кода)

4.2. Управление на паметта и алгоритми. Оптимизации.

4.3. Планиране, описание и създаване на тестови сценарии (създаване на примери)

- 5. Заключение

5.1. Обобщение на изпълнението на началните цели

5.2. Насоки за бъдещо развитие и усъвършенстване

2. Преглед на предметната област

2.1. Основни дефиниции, концепции и алгоритми, които ще бъдат използвани

JSON или *JavaScript Object Notation* е популярен формат за съхранение на данни в дървовиден вид. Често се използва за изпращане на информация през мрежата, съхранение на конфигурационни файлове *(configs)* и много други. Лесно може да се разчете без специален софтуер, но има стриктна структура и правила, за които е най-добре да се използва помощна програма, за да се гарантира, че направените промени няма да развалят структурата на данните. Файловете могат да съхраняват единична стойност, обект или масив от данни. Всеки обект има двойка ключ-стойност, като ключът е от тип string, а стойността може да бъде число, дума, изречение, булева стойност, друг обект, масив или празна стойност. За повече информация, сайтът *json.org* предлага добри обяснения и графики. Алгоритъмът на програмата ще използва рекурсивно извикване за създаване на дървовидната структура от данни, а за четене и търсене на стойности ще се използва търсене в дълбочина.

2.2. Дефиниране на проблеми и сложност на поставената задача

Проблемът, който трябва да се реши, включва взимане на даден файл, анализиране с цел разделяне на данните и запазването им в някаква структура, която да позволява лесна обработка и извеждане на данните. Обработката на входния файл е лесна за разбиране, но се оказа голямо предизвикателство за имплементиране. Тестването на системата се оказа значително по-трудно от първоначално предвидената замисъл.

2.3. Подходи, методи (евентуално модели и стандарти) за решаване на поставените проблемите.

За работа с проекта беше избран команден интерфейс *(Command Line Interface*). За решението на дадения проблем беше създаден главен клас *JsonParser* и няколко отделни класа за съхранение на информацията. Беше създаден валидатор, който минава през данните, ги съхранява и валидира кода,

като се вземат предвид важни елементи от документацията. Също така бяха добавени функции за работа с файловата система. Функцията минава през обекта, гледа за важни елементи и създава дървовидната структура. За четене на данните се използва търсене в дълбочина, след което се запазват и при нужда се манипулират на даденото място.

2.4 Потребителски (функционални) изисквания (права, роли, статуси, диаграми, ...) и качествени (нефункционални) изисквания (скалируемост, поддръжка, ...)

За работа с програмата не е необходим администраторски достъп или някакви допълнителни специални права, но програмата трябва да има достъп до четене и писане на компютъра. Всички команди могат да се извикват след като програмата е стартирана. Голяма част от кода може да се използва и в други програми, но някои функционалности (като някои от извеждането на грешки) са директно вградени в основните класове и може да бъдат модифицирани от кода.

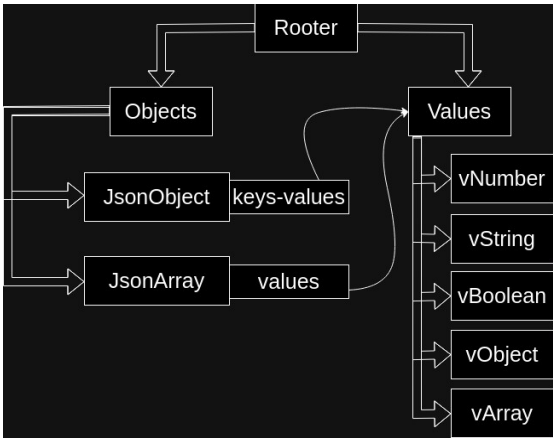
3. Проектиране

3.1. Обща архитектура – ООП дизайн

Има няколко главни класа - „Rooter“, „Objects“, „Values“, като те са организирани така, че да могат да се извеждат подкласовете чрез абстракция. „Objects“ има два подкласа - „JsonObject“ и „JsonArray“, а за „Values“ са изведени всички възможни стойности, които „JsonObject“ и „JsonArray“ могат да съхраняват. Това са „vNumber“, „vString“, „vBoolean“, „vArray“, „vObject“. Класовете „Objects“ и „Values“ имат различни абстрактни функции, което налага класовете да бъдат разделени. Същия проблем се среща и между „JsonObject“ и „JsonArray“.

Понеже класа „Values“ се използва да се напише една обща функция `void* getData(unsigned int) override`, която да бъде прехвърлена на подкласовете му, то тази функция връща „празен“ показател, който после след проверка на типа на класа чрез `ValueType typeOfObject(Objects* obj)` може да се провери типа на дадения обект и да се обработи. Причината за този дизайн е да не се връща обща структура, която да има няколко различни типа данни в нея и само един от тях да съхранява информация.

3.2. Диаграми (на структура и поведение - по обекти, слоеве с най-важните извадки от кода)



Диагра 1. структура на класовете

```
You, 2 days ago | 1 author (You)
class Values : public Rooter{
    public:
    Values() {};
    virtual Values* Clone() = 0;
    virtual int setData(void*) =0 ;
    virtual int removeData(unsigned int) = 0;
    virtual void* getData(unsigned int index = 0) = 0;
    virtual ~Values() {};
};
You, 2 days ago | 1 author (You)
```

```
You, 2 days ago | 1 author (You)
class Objects : public Rooter{

    protected:
    size_t size;
    virtual void expand() = 0;
    virtual void shrink() = 0;
    public:
    size_t Size();
    virtual Objects* Clone() = 0;
    virtual ~Objects() {}
};
```

Снимка 2 и 3. Абстрактните родителски класове

4.Реализация, тестване

4.1. Реализация на класове (включва важни моменти от реализацията на класовете и малки фрагменти от кода)

Всеки клас има динамична памет, като това представлява показател към стойност, за класовете от тип „Values“, а при класовете от тип „Objects“ това е показател от показатели, като идеята е да се инициализира масив от показатели към други обекти (двойки ключ-стойност за „JsonObjects“и стойности за „JsonArray“). При това беше създадена функция клониране, която да връща копие на сегашния елемент. Бяха създадени виртуални деструктори за ичистване на динамичната памет, която се използваше навсякъде.

```
You, 2 days ago | 1 author (You)
class jsonObject : public Objects{
    protected:
    Pair** pairs;

    void expand() override;
    void shrink() override;

    public:
    jsonObject();
    jsonObject(jsonObject&);
    Objects* Clone() override ;

    void AddPair(Pair * pair);
    void AddPair(std::string key, Values* value);

    Pair* ReturnPair(std::string key);
    Pair* ReturnPair(unsigned int index);

    int RemovePair(std::string key);
    int RemovePair(unsigned int index);
    ~jsonObject() override;
};
```

Снимка 4.
Обектен клас съхраняващ двойка “ключ-стойност”

```
class vString : public Values{
    protected:
    std::string *value;
    public:

    Values* Clone() override;
    int setData(void*) override ;
    int removeData(unsigned int) override;
    void* getData(unsigned int) override;

    vString();
    vString(vString&);
    vString(std::string);
    vString(std::string*);
    ~vString() override;
};
```

Снимка 5.
Стойностен клас Съхраняващ дума, изречение или синвули

```
struct Pair{
    std::string key;
    Values *value;
    Pair* Clone(){
        if(value == nullptr){
            return new Pair(key: "", value: nullptr);
        }
        return new Pair(key: this->key, value: value->Clone());
    }
    Pair(){

    }Pair(Pair & p){
        this->key  =p.key;
        this->value = p.value;
    }
    Pair(std::string key, Values* value){
        this->key = key;
        this->value = value;
    }
    ~Pair(){
        delete this->value;
    }
};
```

Снимка 6.
Структура Pair използвана при JsonObject

4.2. Управление на паметта и алгоритми. Оптимизации.

Алгоритъма за обработка на данните е сравнително дълъг и беше пренаписан няколко пъти, за да покрива различни невалидни стойности и за по-лесна работа с него при нуждата от подобрения. Главната му идея е че минава през данните като има няколко ключови елемнта като скоба за нов елемент или масив („ {, }, [,] „), двоеточие за намиране на следващ елемент („ : „) , или запетая на нов елемент („ , „). При намиране на скоба трябва да се извика функцията рекурсивно от момента до където е стигнала сега и да се продължи обработката на данните там, като това което тази функция върне ще е нов елемент в досегашния ключ. Когато функцията се върне, тя ще изгнорира следващите елементи, когато не намери затварящата скоба на сегашната отворена и така ще продължи докато не стигне до края на функцията. Ако рекурсивно извиканта функция намери грешка, тя ще върне стойност false и процеса на валиция ще спре. Скоростa не мога да изчисля, понеже той е рекурсивен и зависи от елементите в данните, но смятам, че най-лошия вариант е O(n + k*m), където n е входните данни, k размера на най-големия елемент (започвайки от отворена скоба до затварящата я) и m – колко подобни елемнти се

срещат. Друг начин за имплементиране на този алгоритъм като идея беше рекурсивно извикване, но когато се върне да върне до кой елемент е стигнала и оттам да продължи главната инициализация на функцията, което щеше да е много по-бърз алгоритъм със скорост O(n), където n е броя на елемнтите в файла.

4.3. Планиране, описание и създаване на тестови сценарии (създаване на примери)

Тестовете, които бяха направени имаха за цел да тестват дали обектите от различните класове правилно съхраняват информация, дали могат да я обработват и да я връщат.

Тук беше намерен проблем, при които в някои тестове при изтриване на показатели, други странични тестове биват повлиявани. При премахването на показателите или на целия тест, другите тестове тръгваха.

5. Заключение

5.1. Обобщение на изпълнението на началните цели

Повечето имплементаци бяха направени. Има нужните класове и те работят. Четенето от файловете също е без проблемно, както и записването в тях (беше имплементирана функция print за лесно извеждане на данните). За жалост не беше имплементиранта опцията валидатора да засича масиви, въпреки че функциалностите нужни за това, класове и алгоритми, са там. Също беше намерен проблем при четенето на данните, който е труден за репликиране. Понякога програмата валидира данните без проблем, а друг път извежда грешка за невалидно заличаване на показатели. Поради някаква причина валидирането на единствена стойност от файл отнема много време.

5.2. Насоки за бъдещо развитие и усъвършенстване

Възможни промени биха били:

1. Подобряване скоростта на алгоритъма
2. Имплементиране на работа с масиви
3. Подобряване на интефейса
4. Динамично извеждане на грешките (да не са вградени в програмта)
5. Подобряване на структурата на класовете

Github : https://github.com/eGuardianDev/JsonParse
За използвани други ресурси повери : https://github.com/eGuardianDev/JsonParse/blob/main/README.md
За намиране на правописни грешки в документацията беше изпозлван *ChatGPT*.