Курсов проект

Обектно-ориентирано програмиране

Кристиян Стоименов
ф.н 3MI0400121, ФМИ

летен семестър, 2023

1 Увод

1.1 Цел

Целта на настоящата курсова работа е бъде реализирана програма, която да обработва и модифицира данни, подредени според разпространения текстов формат *JSON* ("ECMA-404, The JSON data interchange syntax, 2nd edition"). Сред основните функционалности, които трябва да притежава реализираната програма е прочитане на вход и създаване на структура в паметта, следваща формата на подадените данни (т.нар *parsing*), ако те са коректни. В противен случай, ако наложения формат не е спазен, то трябва грешката да бъде докладвана по начин, който предполага лесно да бъде забелязан проблемът в подадения вход. Другата основна част от функционалностите е възможността за подаване на различни команди върху вече прочетен JSON файл. Сред поддържаните команди са операции като вмъкване, премахване, извличане на стойности. Списък с всички необходими команди, както и очакваното от тях поведение, може да се намери в *заданието на проекта* (Тема №6 JSON парсер).

2 Преглед на предметната област

JSON форматът е широко разпространен и намира приложение като начин за съхраняване на *прости* данни. В него данните са наредени предимно като чифтове *ключ-стойност*. В днешно време почти всеки програмен инструмент има някаква връзка с този формат данни било то възможност за негово прочитане, използване като интерпрограмен формат или създаване на изход чрез него. В случая с езика C++ според популярното хранилище за библиотеки "Conan Centre"(Conan Centre) библиотеката `nlohmann_json` (Lohmann) притежава над 400 000 отделни употреби (включва единствено случаите, когато е използвана чрез `conan` (The Conan Package Manager)).

При реализацията на курсовия проект е използвана често срещаната архитектура при обработката и прочитане на текст, който следва фиксираната *форма*. Тази форма обичайно се нарича граматика. Съществуват редица библиотеки, които се използват за съставяне на т.нар *синтактично дърво* от подадена граматика, като тя се описва на отделен език, свързан конкретно с тази задача. Такива са например т.нар *parser generators* yacc (Yet Another Compiler-Compiler) и малко по-новия от него bison (GNU Bison). Те от своя страна разчитат на друга сходна библиотека, която да опише отделните лексеми, които могат да се срещат сред текстовите структури на файловия формат. Такива са т.нар *lexer generators* например flex (flex) и re2c (flex).

3 Проектиране

3.1 Архитектура на проекта

Функционалностите на проекта са разделени в два основни компонента: библиотеката *json-parser* и приложението *json-editor*. Библиотеката се грижи за прочитане на входни данни и създаване на структура в паметта, която да бъде семантична репрезентация на прочетеното, стига то да бъде

валиден JSON. Ако подаденото на входа не е от валиден формат, библиотеката се грижи за откриването на грешката и рапортуването ѝ. Приложението използва интерфейса на библиотеката, така че да реализира поредов редактор.

3.2 Архитектура на библиотеката

3.2.1 Tokenizer

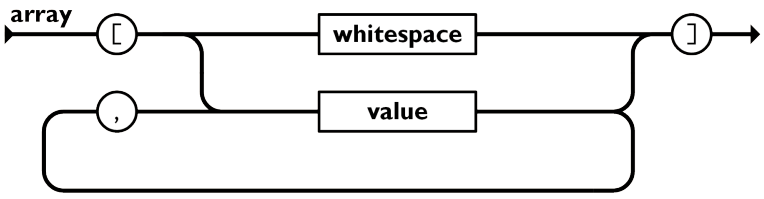
Какво е "tokenizer"? За жалост авторът на настоящия документ не е запознат в утвърден превод на термина "tokenizer поради което това е именно думата, която се среща в описанието на концепцията за *обект*, реализиращ синтактичен анализ върху входни данни от някакъв формат (в случая JSON). Този обект борави с обекти от тип *лексема*, които представляват отделени езикови единици. Това са тези части от свързан текст, които са основополагащи са по-абстрактни и по-сложни обединения в контекста на текстообработката. Често срещани лексеми са например литерали - текстови (стрингове) или числени, пунктуатори и ключови думи. JSON форматът включва няколко прости лексеми, като настоящата разбработка ги третира в четири техни вариации - *число*, *стринг*, *пунктуатор* и *ключова дума*. Основните стъпки, за които се грижи един *tokenizer* включват последователно прочитане на входа и оформяне на прочетеното спрямо очаквания *синтаксис* за лексемите. Фиг. 1 илюстрира изход на типа `tokenizer`, имплементиран в библиотеката *json-parser*, при подаване на вход `{ "apple" : 3.14 }`.

Фигура 1: Изход след изпълнение на *tokenizer*

Конкретната имплементация е разгледана по-подробно в подглава 4.1, посветена на `json-parser` библиотеката.

3.2.2 Parser

Какво е "parser"? За жалост, подобно на случая с 3.2.1, и тук няма да по-добра българска дума, с която до се опише терминът *parsing*. В традиционните текстообработващи системи parser е подсистемата, която се грижи за оформяне на по-абстрактни синтактични струкури, вземайки като вход потокът от лексеми, които предоставя *tokenizer*. Обикновено това се случва чрез използване на *граматика*, която описва какъв е синтаксисът на валиден формат данни. В конкретиката на JSON, отделните граматични елементи са *стринг*, *число*, *масив*, *обект*, *булева стойност* и *празна стойност*. Граматиката на JSON е сравнително просто, което позволява просто имплементация на parser. Най-сложни от нея се явяват структурите масив и обект, които включват в себе си повече от една лексема. Фиг. 2 показва граматиката на масив.



Фигура 2: Диаграма на граматиката на JSON масив (json.org)

3.2.3 Синтактично дърво

След като бъде изпълнене процесът по *parsing*, който от своя страна извиква многократно предоставения нему *tokenizer*, като изход се конструира синтактично дърво. То представлява дървовидна структура, която обхваща всички синтактични елементи, срещнати в подадените входни данни. Тъй като JSON е относително прост формат, сериализиране на неговото синтактично

дърво е практически същото като принтиране в JSON формат (дотолкова сходно, че JSON е популярен формат за принтиране на синтактични дървета на по-сложни езици (-ast-dump=json option in clang)).

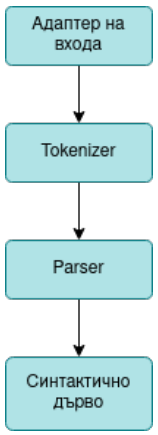
4 Реализация

4.1 Библиотеката *json-parser*

Библиотеката *json-parser* е имплементирана в директорията `lib/`. Тя се компилира като статична библиотека, която в последствие трябва да бъде свързана изрично от изпълнимите файлове, които се нуждаят от нея - това включва тестовете и приложението `json-editor`. Библиотеката реализира обработване (*parsing*) на JSON вход от различни източници. Като своеобразен изход от нея се явява типът `json`, който представлява структура в паметта на съдържанието на входните данни, което позволява изпълнение на различни операции.

Основните компоненти се базират на нивото на абстракция, от което разглеждаме посочените данни. Те са обхванати от типове, които биват използвани от следващите нива и *стъпки от parsing процеса*. Фигура 3 илюстрира стъпките, които са небходими при създаване на синтактичното дърво. Ще разгледаме типовете данни, които се използват поотделно.

Адаптер на входа. Това е първата стъпка от извличането на синтактично дърво. Типовете, които се използват за това са два - `ifs_input_reader` и `str_input_reader` (намират се в `json-parser/include/input_reader.h`). Те предоставят *общ* интерфейс за прочитане на текст от респективно `std::ifstream` и `std::string`, което означава, че можем да използване тези типа като вход, който да съдържа валиден JSON и да може да бъде разпознат впоследствие от типа `tokenizer`. Всъщност и двата адаптера наследяват общ интерфейс `input_reader`, следвайки принципа на CRTP (Curiously Recurring Template Pattern). Основните функции, които имплементират са свързани с извличане на текст - `peek()`, `get()`, `tell()`, `seek()` и пр. Този слой е използван по-късно при реализацията на приложението *json-editor*, за да се прочитат от команден ред данни, които са във валиден JSON формат.



Фигура 3: Стъпки на *parsing*

Tokenizer. Този тип се грижи за прочитане на входа, използвайки подаден адаптер, като извлича данни от типа `token`, който представлява лексема. Той е абстрактен и се явява базов за по-конкретните лексеми, които могат да се срещнат в JSON данни. Тези лексеми са типовете `token_keyword`, `token_number`, `token_string`, `token_punct`. `tokenizer` е с единствена цел да създава инстанции от итератор `token_citerator`, отговарящ на изисквания на концепцията `std::forward_iterator`. В него се имплементират основните функции, които да прочитат лексеми - това са функциите в семейството на `consume_*()`. Те са частни са класа, като по време на *parsing* се използва интерфейса на итератора.

Parser. В това ниво на абстракция, типът `parser` притежава своя инстанция на `tokenizer`, която използва, за да прочита последстователно лексеми от входа, докато конструктира синтактично дърво. То е представено от типа `json`, поради вече споменатото свойство на JSON формата. Основната част от работата на `parser` типа се извършва от функциите в семейството на `parse_*()`.

Синтактично дърво. На този етап данните ни се представят от типа `json`, като те вече са преминали целия процес на *parsing* и са във вече потвърдено валидно състояние. Това позволява извършване на различни операции върху тях, като двете най-комплексни сред тях са проследяване на път в дървото и извличане на стойности, отговарящи на определено изискване. Те се имплементират респективно от `follow()`, който използва типа `path`, и `extract_mapped_if()`, който получава предикат, на чиято база да се извърши филтрирането на съхраняваните данни.

4.2 Приложението *json-editor*

Приложението *json-editor* е имплементирано в директорията `app`. То се компилира като отделен изпълним файл `json-editor` и се намира в `${output}/app/json-editor/$`, където `${output}` е зададената изходна директория. То реализира поредов редактор за файлове с JSON формат, като се базира на функционалностите предоставени от библиотеката `json-parser`.

Основните компоненти на приложението са класът `editor` и няколко свободни функции, които имплементират съответната функционалност на поддържаните команди. В `editor` се съдържа таблица от техните имена, като всяка една от тях е свързана с указател, сочещ към свободна функция, която изпълнява операцията, съответстваща на името. Класът има членове-функции, които добавят или премахват елементи от този вид. След като са добавени всички команди, се извиква `loop`, която се грижи да последователно прочитане на вход и изпълнение на посочената команда със съответните аргументи. Входът на всяка команда е различен, което означава, че свободните функции сами се грижат са снабдяване с небходимите аргументи.

Следното парче код отговаря за създаване на такъв *ректор*, който да отговаря на командите `"help"`, `"open"` и `"close"` (*из app/src/main.cpp*).

```
editor cmdline { "Greetings!" };
cmdline.add_cmd("help", commands::help_cmd);
cmdline.add_cmd("open", commands::open_cmd);
cmdline.add_cmd("close", commands::close_cmd);
// [...]
cmdline.loop();
```

Файлът `app/src/main.cpp` създава инстанция на класа `editor`, която може да изпълнява небходимите команди, посочени в заданието, а `app/src/editor.h` и `app/src/editor.cpp` имплементират типа `editor` и свободните функции, които да изпълняват съотвените команди.

4.3 Библиотеката *mystd*

Библиотеката `mystd` е имплементирана в директориите под `mystd/` и има тестове в `tests/mystd/`. Основната ѝ цел е да предостави някои типове и функции от стандартната библиотека (**stl**), които са използвани в реализацията на курсовия проект, но според заданието това не е разрешено. Това са типовете `unique_ptr`, `unordered_map` и `optional`, както и някои помощни функции за работа с тях като `make_unique`. Тяхната имплементация живее в именованото пространство `mystd`.

Важно е да се отбележи, че ефикасността на реализацията на тези типове не е приоритет на настоящия курсов проект. Същото важи и за придържането към стандарта на езика C++ (Working Draft, Standard for Programming Language C++). Това е използвано като имплементацията на `optional` се свежда до `unique_ptr`, а тази на `unordered_map` до две член-промеливи от тип `std::vector`.

В същото място, където се помещават имплементациите на типовете от `mystd` се намира и файл `enable.h`, който използва макро като маркер относно това дали курсовият проект да използва точно типовете от *"моята стандартна библиотека"* или същинската такава. Тази функционалност на *feature toggle* бе редовно използвана при реализацията на основните компоненти на проекта.

4.4 Тестване

По време на работата по проекта бе използвана библиотеката на съставяне на функционални тестове още наричани *unit tests* `gtest` (googletest). Създадени са тестови случаи за всички компоненти на библиотеката `json-parser`, като се намират различни такива за отделните стъпи по процеса, който се изпълнява при *parsing* на текстов вход. Освен тях е използван и подходът, забелязан в тестове на популярната библиотека `nlohmann_json` (Lohmann), а именно имплементираният в `test_reprint`. Той предполага изпълнение на следния процес:

1. Прочитане на валиден JSON вход и създаване на `json` обект в паметта.
2. Сериализиране на извлечения обект в отделен файл.

3. Повторно прочитане на валиден JSON вход, но този път от туко-що създадения файл, и създаване на `json` обект в паметта.

4. Сериализиране на извлечения обект в отделен файл.

5. Сравняване на двата изходни файла. Очакваме те да бъдат с идентично съдържание.

В предоставения набор от тестови случаи се срещат доста такива, които очакват да се засече грешка в подадения текст. Най-много такива се виждат във файла `tests/test_parser.cpp`, като там са вписани и получените грешки.

Освен за библиотеката `json-parser`, предоставени са тестове и за `mystd`. Те също са изцяло функционални, като се базират на примерите, предоставени в документацията на техните съответни стандартни типове.

5 Заключение

Основните цели, които са поставени пред курсовия проект, са изпълнени според посоченото задание. Въпреки това съществуват доста възможности за бъдещо подобрение на настоящата имплементация. Тук са посочени само някои от тях.

Най-просто реализируемата промяна, която би донесла ползи, е изключването на макросите, които са `mystd/enable.h` файла. Това би довело да използване на типовете и функциите, предоставени в стандартната библиотека, които несъмненое изпълняват задачите си по-ефикасен начин и имат несравнимо по-високо ниво на сигурност от гледна точка наличие на уязвимости и грешки.

Основният фронт за подобрения е предоставеният интерфейс за типа `json`. Към момента той е тромав и грозен, като за да се достъпи стойността в конкретен обект е небходимо използване на изричен *cast* от вида `dynamic_cast`. Несъмнено по-приятно и удобно за използване би бил конвенционалният метод с изброяване на инденксни операции, които от своя страна индексират или масив (`json::array`), или обект (`json::object`). Такъв например може да се види в широко разпространената библиотека `nlohmann_json`.

Относно верифицарането на качество най-важното допълнение към настоящия обем тестови случаи е добавяне на т.нар *fuzzy testing* (Fuzzy testing). Това е стандартен тип проверка, която се е доказала като ефективен подход за борба с проблеми свързани с текстообработка, като ежегодно се използва за намиране на проблеми в софтуер, използван от много години. За целта може да бъде използвана популярната библиотека (libFuzzer). Това обаче ще наложи компилиране на проекта, използвайки компилатора `clang` (**llvm**), докато по време на разработка бе използван `gcc` (gcc).

Литература

"-ast-dump=json option in clang", reviews.llvm.org/D60910.
"Conan Centre", conan.io/center.
"Curiously Recurring Template Pattern", en.cppreference.com/w/cpp/language/crtp.
"ECMA-404, The JSON data interchange syntax, 2nd edition". 2017.
"flex", ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html.
"flex", re2c.org/.
"Fuzzy testing", en.wikipedia.org/wiki/Fuzzing.
"gcc", gcc.gnu.org/.
"GNU Bison", www.gnu.org/software/bison/.
"googletest", github.com/google/googletest.
"json.org", json.org.
"libFuzzer", llvm.org/docs/LibFuzzer.html.
Lohmann, Niels. "JSON for Modern C++ parser and generator." github.com/nlohmann/json.
"The Conan Package Manager", conan.io/.
"Working Draft, Standard for Programming Language C++", www.eel.is/c++draft/.
"Yet Another Compiler-Compiler", www.gnu.org/software/bison/manual/html_node/Yacc.html.

"Тема №6 JSON парсер", 2023, docs.google.com/document/d/1yGwTjf8gskWtwMzavdfM4g3cZAkf-ZNiJQaht1i083o/edit?usp=sharing.