Regularni izrazi u C++-u

Ribarić, Marijan

Sveučilište u Rijeci – Odjel za informatiku

Preddiplomski jednopredmetni studij informatike

Marijan Ribarić

Regularni izrazi u C++-u

Završni rad

Mentor: Doc. dr. sc. Marija Brkić Bakarić

Rijeka, 27.8.2018.

Sadržaj

Zadatak

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 13.3.2018.

Zadatak za završni rad

Pristupnik: Marijan Ribarić

Naziv završnog rada: Regularni izrazi u C++-u

Naziv završnog rada na eng. jeziku: Regular expressions in C++

Sadržaj zadatka:

Regularni izrazi standardan su način za pronalaženje uzoraka u tekstovima. Zadatak rada je dati uvod u regularne izraze općenito, a potom prikazati podršku koju standardna C++ biblioteka ima za regularne izraze i kroz brojne ilustrativne primjere prikazati rad s regularnim izrazima.

Mentor

Doc. dr. sc. Marija Brkić Bakarić

Voditelj za završne radove

Dr. sc. Miran Pobar

Zadatak preuzet: 20.3.2018.

(potpis pristupnika)

1. Sažetak

Regularni izrazi su alat koji nam služi za manipuliranje teksta. U ovom radu objašnjena je sintaksa regularnih izraza s naglaskom na gramatiku *ECMAScript*, kao i metaznakovi koji se koriste pri definiranju regularnih izraza.

Objašnjen je postupak korištenja regularnih izraza u C++-u korištenjem biblioteke *regex*. Kroz primjere je objašnjeno korištenje klase *std::basic_regex<>*, te njezinih specijalizacija kako bi se u C++-u definirali regularni izrazi.

Također su opisane su klase, funkcije i iteratori koje nam pruža biblioteka *regex*, te je kroz primjere objašnjeno njihovo korištenje za pretraživanje i zamjenu teksta.

Ključne riječi: regularni izrazi, C++, ECMAScript, biblioteka regex, pretraživanje teksta, zamjena teksta, pretraživanje datoteka.

2. Uvod

Manipuliranje tekstom čest je zadatak s kojim se susreću programeri. Dok za strukturirane datoteke (npr. JSON, CSV) često postoje parseri koji pružaju jednostavno sučelje za manipuliranje tim datotekama, nestrukturirane datoteke mogu predstavljati problem. Jedno od rješenja tog problema je napisati program koji provjerava znak po znak koristeći petlje i grananje, ali takav program može brzo postati predug i previše kompliciran. Također, za svaki zadatak bilo bi potrebno pisati sav kod ispočetka (Forta, 2018). Jedna od alternativa tome je korištenje regularnih izraza.

Regularni izrazi (engl. *regular expressions*) su nizovi znakova s posebnom sintaksom koji se koriste za pronalaženje i manipuliranje tekstom (Forta, 2018). Regularni izrazi nisu programski jezik već se koriste u sklopu nekog drugog programskoj jezika, kao dio neke aplikacije poput Microsoft Word-a ili alata komandne linije poput *findstr* (Watt, 2005).

Jezik C++ ima službenu podršku za regularne izraze od verzije C++11 u biblioteci *regex* koja je dio standardnog imeničnog prostora (engl. *namespace*) – *std*. Prije objave standarda C++11 *regex* biblioteka je bila dostupna kao dio TR1 (C++ Technical Report 1) nadogradnje (Goyvaerts, 2017).

3. Sintaksa regularnih izraza

Postoji više različitih gramatika koje opisuju sintaksu regularnih izraza. Standardna C++ biblioteka za regularne izraze nudi podršku za šest gramatika (Josuttis, 2012):

- *ECMAScript*
- *basic*
- *extended*
- *awk*
- *grep*
- *egrep*

Sve navedene gramatike počivaju na istim temeljima, dok su glavne razlike u tome koje značajke podržavaju. Na primjer, znak ? u gramatikama *ECMAScript, extended, awk i grep* predstavlja metaznak[1], dok je u gramatici *basic* i *grep* to doslovan znak (engl. *literal*). Gramatika *ECMAScript* je, od navedenih, najpotpunija gramatika, te je također i zadana gramatika u standardnoj C++ biblioteci za regularne izraze (Josuttis, 2012), pa ću se u danjem tekstu fokusirati na nju. Valja napomenuti kako *ECMAScript* gramatika u C++-u nije identična *ECMAScript* gramatici definiranoj u *ECMA-262v3* i *POSIX* standardu, već je u manjoj mjeri modificirana (Goyvaerts, 2017).

Regularni izrazi sastoje se od doslovnih znakova i metaznakova (Forta, 2018). Regularni izraz koji se sastoji od znamenke ili slova (doslovni znakovi) upariti (engl. *match*) će se sa pojavljivanjem te znamenke ili slova u tekstu kojeg pretražujemo (Watt, 2005). Isto vrijedi i za niz znakova. Ukoliko je znak koji želimo pronaći metaznak potrebno je izbjeći njegovo posebno značenje korištenjem znaka \, takozvanog znaka za izbjegavanje (engl. *escape character*)[2]. Znak . (točka) je metaznak koji predstavlja bilo koji znak koji nije znak za novu liniju. To uključuje mala i velika slova iz engleske abecede, znamenke, kao i veliku većinu ne-engleskih slova (Watt, 2005).

Regularni izrazi su osjetljivi na veličinu slova. Međutim, većina alata nudi opcije kojima regularni izrazi ignoriraju veličinu slova (Forta, 2018).

3.1. Klase znakova

Klasa znakova je neuređena grupa znakova iz koje se odabire jedan od znakova za uparivanje pomoću regularnog izraza (Watt, 2005). Označava se korištenjem znaka [(otvorena uglata zagrada) za početak klase, te znaka] (zatvorena uglata zagrada) za kraj. Unutar klase može stajati jedan ili više doslovnih znakova. Regularni izraz „1[123]4“ primjer je korištenja klasa. Ovaj regularni izraz uparuje se s brojevima 114, 124 i 134.

Osim doslovnih znakova unutar klase se može nalaziti i opseg znakova. Ukoliko se znak - (crtica) nalazi između dva doslovna znaka unutar klase on postaje metaznak. U tom slučaju znak - označava opseg znakova između znaka koji mu prethodi, te znaka koji mu slijedi[3]. To znači da su u klasi obuhvaćeni navedeni znakovi te svi znakovi po ASCII (engl.

[1] Metaznak je znak koji prenosi značenje različito od svojeg običnog značenja (Watt, 2005). U ovom slučaju znak ? označava da se prethodni izraz ponavlja 0 ili 1 put.

[2] Znak „\“ je također metaznak, pa ako bismo htjeli pronaći znak „\“ regularni izraz bi glasio „\\“.

[3] U ostalim slučajevima znak „-“ se smatra doslovnim znakom.

American Standard Code for Information Interchange) tablici između njih. Neki od primjera regularnih izraza s opsegom unutar klase su (Forta, 2018):

- „[A-Z]“ – sva velika slova između A i Z
- „[a-z]“ – sva mala slova između a i z
- „[A-F]“ – sva velika slova između A i F
- „[0-9]“ – sve dekadske znamenke

Opseg znakova [A-z] također je važeći, ali se njegovo korištenje ne preporuča zato jer se u ASCII tablici između znakova A i z nalaze i neki znakovi koji nisu slova poput ^ i [(Forta, 2018). Umjesto toga može se koristiti regularni izraz „[a-zA-Z]“.

Znak ^ postaje metaznak ukoliko se nalazi neposredno iza znaka [koji označava početak klase. Tada on signalizira da se uparuju svi znakovi osim onih u klasi. Ukoliko se znak ^ nalazi na nekom drugom mjestu u klasi smatra se doslovnim znakom.

3.1.1. Prečaci za klase znakova

Kako bi se izbjeglo pisanje cijelih klasa implementirani su prečaci za često korištene klase znakova:

- \d – predstavlja sve znamenke od 0 do 9
- \D – predstavlja sve osim znamenaka od 0 do 9
- \w – predstavlja sve znakove iz engleske abecede, znamenke i znak _
- \W – predstavlja sve što ne spada pod \w
- \s – predstavlja sve znakove bjeline, tj. razmak, znak \n i znak \t
- \S – predstavlja sve što ne spada pod \s

Ove klase podržava, od gramatika implementiranih u C++-u, jedino *ECMAScript* gramatika (Josuttis, 2012).

3.1.2. POSIX klase

POSIX klase su još jedan od prečaca za klase znakova. Te klase imaju sintaksu oblika [:imeklase:], te se pišu unutar obične klase znakova. *POSIX* klase nisu podržane u svim gramatikama. To vrijedi i za standardnu verziju *ECMAScript* gramatike. Međutim, proširena verzija *ECMAScript* gramatike koju C++ implementira podržava *POSIX* klase (Goyvaerts, 2017).

Tablica 1. POSIX klase i njihov opis (Forta, 2018)

Oznaka klase	Opis
[:alnum:]	Bilo koje slovo ili znamenka
[:alpha:]	Bilo koje slovo
[:blank:]	Razmak ili tabulator
[:cntrl:]	ASCII kontrolni znak
[:digit:]	Bilo koja znamenka
[:graph:]	Bilo koji znak koji se može ispisati (osim razmaka)
[:lower:]	Bilo koje malo slovo

[:print:]	Bilo koji znak koji se može ispisati
[:punct:]	Bilo koji znak koji nije [:alnum:] ni [:cntrl:]
[:space:]	Bilo koji znak koji uključuje bjeline
[:upper:]	Bilo koje veliko slovo
[:xdigit:]	Bilo koja heksidecimalna znamenka

3.2. Kvantifikatori

Kvantifikatori (ili operatori kardinalnosti) su metaznakovi koji označavaju da se izraz koji im prethodi ponavlja određeni broj puta.

Metaznak * (zvjezdica) označava da se izraz koji prethodi ponavlja nula, jednom ili više puta. Drugim riječima, znak * nam govori da je izraz opcionalan, ali i da se može pojaviti više puta (Watt, 2005). Na primjer, regularni izraz „1\.2*“ bi prihvaćao nizove „1.“, „1.2“, „1.22“, itd.

Metaznak + (plus) označava da se izraz koji prethodi ponavlja jednom ili više puta. Na primjer, regularni izraz „a+“ bi prihvaćao nizove „a“, „aa“, „aaa“, itd.

Metaznak ? (upitnik) označava da se izraz koji prethodi pojavljuje jednom ili nijednom. Na primjer, regularni izraz „ab?“ bi prihvaćao nizove „a“ i „ab“.

Ako želimo precizno zadati broj ponavljanja koristimo metaznakove { i } (vitičaste zagrade) na slijedeći način:

- {n} – prethodni izraz se ponavlja n puta
- {n,m} – prethodni izraz se ponavlja od n do m puta (brojevi n i m su uključeni u interval)
- {n,} – prethodni izraz se ponavlja najmanje n puta

3.3. Grupiranje znakova

Metaznakovi (i) (zagrade) služe za grupiranje dijelova izraza u podizraze. Podizrazi su dijelovi većeg izraza koji su zajedno grupirani, te se tretiraju kao zasebna cjelina (Forta, 2018). Kvantifikatori djeluju na podizraze koji im prethode. Unutar grupe znak | postaje metaznak koji označava da se pri traženju uzorka bira niz znakova lijevo ili desno od metaznaka |. Podizrazi se mogu pojavljivati unutar drugih podizraza.

3.3.1. Capturing group

Grupirani izraz postaje tzv. *capturing group*, te se kasnije može koristiti kao *backreference*. Osim toga, podizrazu se može u C++-u individualno pristupati, te se podizraz može koristiti u uzorku za zamjenu (Gregoire, 2014). Ukoliko želimo spriječiti to ponašanje na određenoj grupi možemo nakon otvorene zagrade dodati znak ?. U tom slučaju se grupirani izraz ne može koristiti na prije navedene načine, ali na njega i dalje djeluju kvantifikatori.

3.3.2. Backreference

Ukoliko unutar regularnog izraza želimo ponovno koristiti isti podizraz koji je pronađen koristimo *backreference*. *Backreference* se označava s \i gdje je *i* broj podizraza.

Primjer korištenja *backreference* uzorka je traženje uzastopnih riječi koje se ponavljaju. Regularni izraz „\d+ \1“ tako pronalazi nizove znakova „ime ime“, „the the“…

3.3.3. Lookahead

Pozitivni *lookahead* uzorak je vrsta podizraza koji nam omogućava da se grupa znakova pronađe bez da se „*potroši*“ (engl. *consume*). Drugim riječima, podizraz se uparuje, ali se pronađeni izraz ne sprema, te se ponovno može pronaći (Forta, 2018). Pozitivni lookahead se označava znakovima ?= nakon zagrade koja označava početak grupe znakova. Negativni *lookahead-om* označavamo uzorak koji se ne smije pojaviti da bi pronađeni uzorak bio valjan, također bez „*trošenja*“ niza znakova koji se uparuju. Negativni lookahead se označava znakovima ?! nakon zagrade koja označava početak grupe znakova (Gregoire, 2014).

Također, postoje i *lookbehind* uzorci koji provjeravaju odgovaraju li znakovi koji prethode regularnom izrazu određenom uzorku. Međutim, *lookbehind* uzorci nisu podržani u C++-u (Goyvaerts, 2017).

3.4. Ostali metaznakovi

Metaznak ^ označava početak linije ili početak niza znakova (engl. *string*), dok metaznak $ označava kraj linije ili kraj niza znakova. Ukoliko naš niz znakova ima više linija, da bi dobili rezultat koji očekujemo, u mnogim alatima (uključujući C++) potrebno je uključiti višelinijski način rada (engl. *multiline mode*) (Watt, 2005). Ova opcija u C++-u postoji od verzije C++17.

Znak za novi red se označava sa \n, dok se znak za tabulator označava sa \t.

Znak \b označava granicu riječi, odnosno znakove koje razdvajaju riječi. Znak \b ne zahtjeva da se pronađe određeni znak, već da se na poziciji na kojoj se nalazi u regularnom izrazu nalazi neki znak koji nije dio riječi (razmak, interpunkcijski znak i slično) (Forta, 2018). Ukoliko želimo da se na određenoj pozicija nalazi znak koji ne označava granicu riječi koristimo znak \B.

4. Definiranje regularnih izraza u C++-u

Službena podrška za regularne izraze nalazi se u biblioteci regex uz pomoću koje možemo koristiti klasu *std::basic_regex<>* i pomoćne funkcije (Stroustrup, 2014). Da bismo ih koristili moramo je uključiti u projekt kao što je prikazano na slici:

```
#include <regex>
```

Slika 1. Isječak koda - Uključivanje biblioteke regex

Za početak rada s regularnim izrazima u C++-u potrebno je inicijalizirati objekt klase predloška *std::basic_regex<>*. Za to imamo dvije mogućnosti (Goyvaerts, 2017):

- Korištenje klase *std::regex* ukoliko je niz znakova s kojim manipuliramo tipa *std::string* ili niz tipa *char*.
- Korištenje klase std::wregex ukoliko je niz znakova s kojim manipuliramo tipa *std::wstring* ili niz tipa *wchar_t*.

Preporuča se korištenje tip podataka *std::string* (Stroustrup, 2014) pa ću se u nastavku fokusirati na klase i funkcije koje koriste taj tip podataka.

Objekt klase *std::regex* se kreira koristeći jedan od konstruktora. Objekt kreiramo na slijedeći način:

```
void kreiranjeObjekta(){
    std::regex re("C\\+\\+");
    std::regex re2("C\\+\\+", std::regex::egrep);
    std::regex re3(re);
}
```

Slika 2. Isječak koda - Načini kreiranja objekta regex

Na slici su prikazana tri načina kreiranja *std::regex* objekta. Prosljeđivanjem samo regularnog izraza, prosljeđivanjem regularnog izraza i opcionalnih zastavica (engl. *flags*), te kopiranjem postojećeg objekta.

4.1. Znak izbjegavanja

Kako se moglo primijetiti iz prethodnog primjera, da bi se dobilo doslovno značenje metaznaka potrebna su dva znaka za izbjegavanje. Razlog tome je to što obrnuta kosa crta specijalni znak u C++ *stringu*, te je potrebno izbjeći posebno značenje tog znaka korištenjem još jedne obrnute kose crte.

Kako bi se olakšalo pisanje i povećala čitljivost koda, preporuča se korištenje „sirovog“ niza znakova (engl. *raw string*). To je vrsta string-a koji dopušta da se obrnuta kosa crta i navodnici koriste direktno u *stringu* (Stroustrup, 2014). Primjer korištenja *raw string-a* može se vidjeti na slici:

```
void demonstracijaEscapeCharactera(){
    std::regex re("C\\+\\+");
    std::regex re2(R"(C\+\+)");
}
```

Slika 3. Isječak koda - Korištenje raw string-a

Prikazani regularni izrazi su identični.

4.2. Regex zastavice

Pri kreiranju objekta tipa *std::basic_regex<>* mogu se proslijediti zastavice koje određuju kakvo će biti ponašanje regularnog izraza. Može se proslijediti više zastavica koristeći operator | (*bitwise or operator)*. Postoje slijedeće zastavice za *std::basic_regex<>* objekt:

- *icase* – Regularni izraz postaje neosjetljiv na veličinu slova
- *nosubs* – Rezultat se neće dijeliti na grupe
- *optimize* – Produžuje se vrijeme kreacije, ali smanjuje vrijeme traženja
- *collate* – Korišteni jezik utječe na opseg kod grupa
- *ECMAScript, basic, extended, awk, grep, egrep* – koristi se istoimena gramatika. Samo jedna od ovih zastavica smije biti prisutna. Ukoliko se ne odabere ni jedna zastavica, zadana vrijednost je *ECMAScript*.

4.3. Iznimke

Ukoliko se dogodi greška tijekom parsiranja regularnog izraza „baca“ se iznimka (engl. *throw exception*) tipa std::regex_error. Klasa std::regex_error, osim metode *what()* (koju nasljeđuje iz *std::exception*), daje i pristup metodi *code()* iz koje možemo dobit kod greške koja se dogodila. Kod greške ovisi o implementaciji, tako da direktno iz njega ne možemo zaključiti o kojoj se grešci radi (Josuttis, 2012). Iz tog razloga potrebno je usporediti kod dobiven iz metode *code()* sa konstantama definiranim u *std::regex_constant::error_type*. Donji primjer pokazuje korištenje iznimaka.

```
void regexExceptions(){
    try {
        std::regex re(R"(a{0,2)");
    }
    catch (const std::regex_error& e) {
        std::cerr << "Error: " << e.what() << '\n';
        if(e.code() == std::regex_constants::error_brace)
            std::cerr<<"Viticaste zagrade nisu pravilno zatvorene";
    }
}
```

Slika 4. Isječak koda - Hvatanje iznimke

Očita je pogreška u regularnom izrazu (nedostatak zatvorene vitičaste zagrade). Nakon što se zbog greške prekine izvođenje programa u bloku try, u bloku catch „hvata“ (engl. *catch*) se „bačena“ iznimka. Iz objekta *std::regex_error* iščitavamo vrstu greške (metoda *what()*), te kod greške uspoređujemo sa konstantom za grešku s vitičastim zagradama (*error_brace*). Pokretanjem programa dobijemo slijedeći rezultat:

```
Error: regex_error
Viticaste zagrade nisu pravilno zatvorene
Process returned 0 (0x0)   execution time : 0.047 s
```

Slika 5. Rezultat programa - Hvatanje iznimke

Ukoliko želimo da nas naš program obavijesti o svim definiranim vrstama greške potrebno je prije prikazanu „ako“ tvrdnju ponoviti za sve vrste grešaka. Takvo izvješćivanje

o greškama preporučeno je implementirati u posebnu funkciju koristeći switch grananje (Josuttis, 2012).

5. Korištenje regularnih izraza u C++-u

Biblioteka *regex* nudi podršku za korištenje regularnih izraza putem funkcija *regex_match()*, *regex_search()* i *regex_replace()*, te iteratora *regex_iterator<>* i *regex_token_iterator<>* (Josuttis, 2012).

5.1. Uparivanje cijelog niza znakova

Za uparivanje cijelog niza znakova koristi se funkcija *regex_match()*. Ona provjerava odgovara li cijeli niz znakova uzorku regularnog izraza (Josuttis, 2012). Funkcija vraća vrijednost *true* ukoliko niz znakova odgovara, te *false* ako vrijedi suprotno. Funkcija ima nekoliko verzija, tj. nadjačavanja (engl. *overload*). Na slici 6 su prikazana dva različita poziva u kojoj se funkciji prosljeđuje ili *string* ili iterator koji pokazuje na početak *stringa* i iterator koji pokazuje na kraj *stringa*, te objekt *std::basic_regex<>* (Josuttis, 2012).

```
void demonstacijaRegexMatch(){
    std::regex re2(R"(C\+\+)", std::regex::icase);
    std::string s1 = "c++";
    std::string s2 = "C+";

    if(std::regex_match(s1, re2))
        std::cout<<"s1 odgovara uzorku\n";
    else
        std::cout<<"s1 ne odgovara uzorku\n";

    if(std::regex_match(s2.begin(), s2.end(), re2))
        std::cout<<"s2 odgovara uzorku\n";
    else
        std::cout<<"s2 ne odgovara uzorku\n";
}
```

Slika 6. Isječak koda - Korištenje funkcije regex_match()

```
s1 odgovara uzorku
s2 ne odgovara uzorku
```

Slika 7. Rezultat programa - Korištenje funkcije regex_match()

5.2. Pronalaženje uzorka u nizu znakova

Ukoliko želimo pronaći uzorak definiran regularnim izrazom u podnizu niza znakova koristimo funkciju *regex_search()*. Ova funkcija također vraća vrijednost *true* ukoliko je uzorak pronađen, te *false* ukoliko nije. Da bi u potpunosti iskoristili mogućnosti ove funkcije (i funkcije *regex_match()*, također) potrebno je spremiti rezultate pretraživanja.

5.2.1. Rezultat pretraživanja

Klasa *std::match_results<>* je klasa predloška koja služi za spremanje rezultata pretraživanja. Postoje četiri predefinirane specijalizacije (Josuttis, 2012):

- *smatch* – za detalje o rezultatima u tipu *string*
- *cmatch* – za detalje o rezultatima u tipu *C-string* (*const char**)
- *wsmatch* – za detalje o rezultatima u tipu *wstring*
- *wcmatch* – za detalje o rezultatima u tipu „široki“[4] *C-string* (*const wchar_t**)

[4] Široki (engl. *wide*) niz znakova je niz znakova koji podržava više znakova od standardnog niza

Potrebno je koristiti isti tip za rezultat kao i onaj koji se koristi za niz koji pretražujemo koristeći funkcije *regex_match()* ili *regex_search()* (Josuttis, 2012).

Primjer korištenja klase *std::smatch*, te nekih od njezinih mogućnosti možemo vidjeti na slijedećem primjeru:

```
void rezultatiPretrazivanja(){
    std::regex re(R"((\w+(?:\.\w+)*)@(\w+\.[[:alpha:]]+))");
    std::smatch rezultat;
    std::string emailovi = "Email: as.df@gmail.com\n"
                           "Email: ivoIvic@net.hr\n"
                           "Email: kriviMail@.hr";
    if(std::regex_search(emailovi, rezultat, re)){
        std::cout<<"Email: "<<rezultat[0]<<"\n";
        std::cout<<"Korisnicko ime: "<<rezultat[1]<<"\n";
        std::cout<<"Domena: "<<rezultat[2]<<"\n";
        std::cout<<"Pozicija rezultata: "<<rezultat.position()<<"\n";
        std::cout<<"Velicina rezultata: "<<rezultat.size()<<"\n";
        std::cout<<"Prefix: "<<rezultat.prefix()<<"\n";
        std::cout<<"Suffix:"<<rezultat.suffix()<<"\n";
    }
}
```

Slika 8. Isječak koda - Rezultat pretraživanja

Nakon što smo definirali regularni izraz za pretraživanje e-mail adresa, te objekte *std::smatch* i *std::string* pozivamo funkciju *regex_search*. Objekt rezultat prosljeđuje se kao referentni parametar u funkciju. Ukoliko je uzorak pronađen ispisuju se atributi. Operator [n] vraća n-tu podgrupu u pronađenom rezultatu kao objekt std::ssub_match. Nulta podgrupa je cijeli rezultat (Stroustrup, 2014). Alternativa operatoru [] je metoda *str(n)* vraća podgrupu kao objekt tipa *string*. Također su (između ostalih) definirane i slijedeće metode:

- *position()* – vraća poziciju rezultata u promatranom string-u
- *size()* – broj podgrupa
- *prefix()* – sve prije pronađenog rezultata
- *suffix()* – sve nakon pronađenog rezultata

Prethodni program daje slijedeći rezultat:

```
Email: as.df@gmail.com
Korisnicko ime: as.df
Domena: gmail.com
Pozicija rezultata: 7
Velicina rezultata: 3
Prefix: Email:
Suffix:
Email: ivoIvic@net.hr
Email: kriviMail@.hr
```

Slika 9. Rezultat programa - Rezultat pretraživanja

Ukoliko proslijedimo objekt *std::smatch* funkciji regex_match dobiti ćemo rezultat istog oblika, ali će (zato jer funkcija *regex_match* uvijek uparuje cijeli *string*) prefiks (engl. *prefix*) i sufiks (engl. *suffix*) biti prazni (Josuttis, 2012).

5.2.2. Pronalaženje svih rezultata

Klasa *std::match_results* daje nam pristup prvom znaku sufiksa koristeći *suffix().first*. Definiramo početak i kraj ulaznog niza znakova, te u petlji nakon svakog pronalaska svakog uzorka postavimo početak niza na *suffix().first*. Tako možemo pretražiti cijeli ulazni niz znakova. Međutim, kako to ne bismo morali raditi ručno, biblioteka *regex* nam na raspolaganje daje predložak iteratora *regex_iterator<>*. Kao i za rezultate pretraživanja postoje četiri predefinirane specijalizacije sa prefiksima s, c, ws ili wc ovisno o tipu podataka koji koristimo za ulazni niz znakova (Josuttis, 2012).

Konstruktor za *regex_iterator<>* prima iterator koji pokazuje na početak ulaznog niza, iterator koji pokazuje na kraj ulaznog niza, te *std::basic_regex<>* objekt. Opcionalno se mogu proslijediti i zastavice za pretraživanje. Konstruktor bez argumenata označava kraj slijeda (engl. *end-of-sequence*) (Stroustrup, 2014).

Na sljedećoj slici (Slika 10) nalazi se modificirani prethodni primjer tako da se sparuju svi rezultati koji odgovaraju uzorku. U *for* petlji inicijaliziramo iterator kojeg svaku iteraciju petlje inkrementiramo sve dok nije jednak *end-of-sequence* iteratoru.

```
void sviRezultati(){
    std::regex re(R"((\w+(?:\.\w+)*)@(\w+\.[[:alpha:]]+))");
    std::string emailovi = "Email: as.df@gmail.com\n"
                           "Email: ivoIvic@net.hr\n"
                           "Email: kriviMail@.hr";

    for(std::sregex_iterator it(emailovi.begin(), emailovi.end(), re);
            it != std::sregex_iterator(); ++it){
        std::cout<<"Email: "<<(*it)[0]<<"\n";
        std::cout<<"Korisnicko ime: "<<(*it)[1]<<"\n";
        std::cout<<"Domena: "<<(*it)[2]<<"\n";
        std::cout<<"------------------------------\n";
    }
}
```

Slika 10. Isječak koda - Pronalaženje svih rezultata

Kod svake inkrementacije iteratora automatski se poziva funkcija *regex_match()*. Unutar petlje se ispisuju rezultati pomoću smatch objekta na koji iterator pokazuje. Pokretanjem programa dobivamo slijedeći izlaz:

```
Email: as.df@gmail.com
Korisnicko ime: as.df
Domena: gmail.com
------------------------------
Email: ivoIvic@net.hr
Korisnicko ime: ivoIvic
Domena: net.hr
------------------------------
```

Slika 11. Rezultat programa - Pronalaženje svih rezultata

5.2.3. Razdvajanje na tokene

Osim iteratora *regex_iterator<>* postoji i iterator *regex_token_iterator<>*. On iterira kroz podgrupe pronađenih rezultata, odnosno kroz objekte tipa *std::sub_match*.

U usporedbi s iteratorom *regex_iterator<>* konstruktor iteratora *regex_token_iterator<>* prima dodatan argument: cjelobrojnu vrijednost ili listu cjelobrojnih vrijednosti koje određuju kako će se iterator ponašati. Može se proslijediti jedna ili više vrijednosti, a ukoliko se ne proslijedi ni jedna vrijednost zadana vrijednost je 0. S obzirom na proslijeđene vrijednosti postiže se slijedeće ponašanje (Josuttis, 2012):

- -1 označava podgrupe između pronađenih uzoraka
- 0 označava sve pronađene uzorke
- Bilo koja druga pozitivna vrijednost n označava n-tu podgrupu u pronađenom uzorku

Vrijednosti se mogu grupirati kod prosljeđivanja. Na primjer, ako želimo koristiti vrijednosti -1 i 1 u funkciju ih prosljeđujemo kao argument {-1,1}.

Posebice je zanimljiva vrijednost -1 koja se može koristiti za razdvajanje niza znakova na tokene (najmanje vrijednosti koje nam imaju neko značenje). Primjer korištenja iteratora za razdvajanje *URL-a* na dijelove možete vidjeti na slici:

```
void tokenizacija(){
    std::regex re(R"([\.:/\?&]+)");
    std::string adresa = "http://www.inf.uniri.hr/hr";

    for(std::sregex_token_iterator it(adresa.begin(), adresa.end(), re, -1);
            it != std::sregex_token_iterator(); ++it){
        std::cout<<(*it)<<"\n";
    }

}
```

Slika 12. Isječak koda - Razdvajanje na tokene

U primjeru je definiran regularni izraz koji pronalazi sve znakove koji razdvajanju dijelove *URL-a*. Nakon toga se u *for* petlji kreira iterator *std::sregex_token_iterator<>*, te mu se prosljeđuje vrijednost -1. Kao rezultat dobivamo razdvojene dijelove *URL-a*:

```
http
www
inf
uniri
hr
hr
```

Slika 13. Rezultat programa - Razdvajanje na tokene

5.3. Zamjena korištenjem regularnih izraza

Osim pretraživanja regularni izrazi mogu služiti i za zamjenu teksta s nekim drugim tekstom. Biblioteka regex nudi podršku za to u vidu funkcije *regex_replace()*. Funkcija *regex_replace()* osim argumenata izvornog niza znakova, te objekta std::basic_regex prima i argument formatiranja. U tom argumentu određujemo čime želimo zamijeniti pronađeni uzorak. Definirani su posebni simboli koji se koriste za formatiranje (Josuttis, 2012):

- $& označava cijeli pronađeni uzorak
- $n označava n-tu pronađenu podgrupu
- $‘ označava prefiks pronađenog uzorka
- $’ označava sufiks pronađenog uzorka

- $$ označava znak $

Također se opcionalno mogu proslijediti i zastavice za zamjenu (Josuttis, 2012):

- *format_default* – koristi se standardna (*ECMAScript*) sintaksa za zamjenu
- *format_sed* – koristi se sintaksa alata sed
- *format_first_only* – zamjenjuje se samo prvo pojavljivanje uzorka
- *format_no_copy* – ne kopiraju se znakovi koji ne odgovaraju uzorku

Na slijedećem primjeru demonstrira se korištenje zamjene kako bi se promijenio datum formata MM/DD/YYYY u datum formata DD.MM.YYYY.

```
void zamjena(){
    std::regex re(R"((\d\d?)/(\d\d?)/(\d{4}))");
    std::string replaceFormat = "$2.$1.$3.\n";
    std::string datum = "11/30/1950"
                        "2/1/2001"
                        "1235/643";
    std::cout<<std::regex_replace(datum, re, replaceFormat,
                                  std::regex_constants::format_no_copy);
}
```

Slika 14. Isječak koda – Zamjena

U primjeru se definira regularni izraz koji odgovara traženom formatu datuma, te format u koji pronađeni uzorak želimo pretvoriti. Zatim pozivamo funkciju s prije definiranim parametrima, te dodajemo zastavicu da se ne kopira tekst koji ne odgovara uzorku. Vrijednost koju vraća funkcija odmah ispisujemo na ekran i dobivamo slijedeći rezultat:

```
30.11.1950.
1.2.2001.
```

Slika 15. Rezultat programa – Zamjena

5.4. Pretraživanje tekstualnih datoteka

Regularni izrazi se često koriste za pronalaženje uzoraka u tekstualnim datotekama. Ovisno o strukturi datoteke, te o uzorku koji tražimo, sadržaj datoteke možemo pretraživati na više načina:

- Spremanjem cijelog sadržaja datoteke u jedan *string*
- Čitanjem datoteke liniju po liniju
- Čitanje datoteke riječ po riječ

Nakon što pročitamo datoteku moramo odabrati jednu od prije opisanih funkcija ili jedan od iteratora. Pretraživanje datoteka možemo vidjeti na slijedećem primjeru u kojem se iz tekstualne datoteke izvlače svi telefonski brojevi:

```
std::ifstream inputFile("test_set.txt");
if (!inputFile){
    std::cout<<"test_set.txt is missing.\n";
    return -1;
}

std::ofstream outputFile("reference_set.txt");
std::string currentLine;
std::regex re(R"(\+[1-9]{1,3} ?(\(\d{2,4}\)|\(0\)\d{1,3}|\d{2,3}))"
              R"((( ?(\d{3,4})){2}|( ?\d{2,3}){3,4})\b)");
const std::sregex_iterator end;

while (std::getline(inputFile, currentLine)){
    for (std::sregex_iterator it(currentLine.begin(), currentLine.end(), re);
        it!=end; ++it){
        outputFile<<(*it)[0]<<"\n";
    }
}
```

Slika 16. Isječak koda - Pretraživanje tekstualnih datoteka

Prvo definiramo regularni izraz koji opisuje format telefonskih brojeva[5]. Zatim se u *while* petlji čita iz datoteke *test_set.txt* liniju po liniju. Svaku pročitanu liniju pretražujemo koristeći *std::sregex_iterator* kako bismo dobili sve rezultate. Svaki rezultat spremamo u datoteku *reference_set.txt*. Obije datoteke, kao i cjelokupni programski kod prijašnjeg primjera priloženi su uz ovaj rad.

[5] Format telefonskih brojeva nije standardiziran. Ovdje definirani regularni izraz prihvaća hrvatske brojeve telefona, te većinu svjetskih brojeva telefona.

6. Zaključak

Zbog svoje jednostavnosti i raširenosti regularni izrazi su i danas važan alat za manipulaciju teksta. Iako se sintaksa može razlikovati od implementacije do implementacije, razlike su često minimalne. Tako se isti (ili djelomično modificirani) regularni izraz može koristiti u različitim alatima (npr. *grep, findstr*) i programskim jezicima (npr. Perl, C++) pod uvjetom da se koristi ista gramatika ili da se regularni izraz prilagodi ciljanoj gramatici.

C++ pruža podršku za regularne izraze kroz biblioteku *regex*. Iako je trebalo čekati do 2011. godine za službenu podršku, regularni izrazi su sada dobro podržani sa klasama i funkcijama iz biblioteke *regex*. Implementacija u C++-u možda nije jednostavna kao u jezicima poput Python-a i Perl-a, ali je intuitivna, te se regularni izrazi (nakon što se pročita dokumentacija) mogu brzo početi koristiti bez velikih problema.

Također, biblioteka *regex* podržava najkorištenije tipove podataka za zapis znakova. To uvelike olakšava korištenje regularnih izraza, te se povećavaju performanse programa jer se eliminiraju nepotrebna pretvaranja iz jednog tipa podataka u drugi.

7. Popis slika

8. Popis literature

Forta, B. (2018). *Learning Regular Expressions.* Boston: Addison-Wesley.

Goyvaerts, J. (30. June 2017). *Regular-Expressions.info*. Dohvaćeno iz https://www.regular-expressions.info/

Gregoire, M. (2014). *Professional C++.* Indianapolis: John Wiley & Sons, Inc.

Josuttis, N. M. (2012). *The C++ Standard Library.* Upper Sadle River: Addison-Wesley.

Stroustrup, B. (2014). *A Tour of C++.* Upper Saddle River: Addison-Wesley.

Watt, A. (2005). *Beginning Regular Expressions.* Indianapolis: Wiley Publishing, Inc.

9. Popis izvora

(n.d.). Preuzeto 15. Kolovoz 2018. iz C++ Reference: https://en.cppreference.com/w/

Friedl, J. E. (2006). *Mastering Regular Expressions.* Sebastopol: O'Reilly Media, Inc.

10. Popis priloga

Uz ovaj rad priložene su i slijedeće datoteke:

- test_set.txt
- reference_set.txt
- pretrazivanjeDatoteka.cpp