C++17 i STL

Wyrażenia regularne

Czym są wyrażenia regularne

- **Wyrażenia regularne** to wzorce, które opisują odpowiednio sformatowane łańcuchy znaków. Wyrażenia regularne mogą określać zbiór pasujących łańcuchów, mogą również wyszczególniać istotne części łańcucha.
- W informatyce teoretycznej wyrażenia regularne są ciągami znaków pozwalającymi opisywać języki regularne. W praktyce znalazły bardzo szerokie zastosowanie, pozwalają bowiem w łatwy sposób opisywać wzorce tekstu, natomiast istniejące algorytmy w efektywny sposób określają, czy podany ciąg znaków pasuje do wzorca lub wyszukują w tekście wystąpienia wzorca.
- Wyrażenia regularne stanowią integralną część narzędzi systemowych takich jak **sed**, **grep**, wielu edytorów tekstu, języków programowania przetwarzających tekst **AWK** i **Perl**, a także są dostępne jako biblioteki we wszystkich współczesnych językach obiektowych, w tym **C++**.

Wyrażenia regularne – historia

- Koncepcję wyrażeń regularnych sformalizował amerykański matematyk Stephen Cole Kleene w latach 50-tych XX wieku.
- Wyrażenia regularne weszły do powszechnego użytku w 1968 roku:
  - dopasowywanie wzorców w edytorze tekstu,
  - analiza leksykalna w kompilatorze.
- Regex'y zostały następnie przyjęte przez szeroką gamę programów, przy czym te wczesne formy zostały ujednolicone w standardzie POSIX.2 w 1992 roku.
- Dodatkowo w 1997 roku Philip Hazel opracował PCRE (Perl Compatible Regular Expressions), który stara się naśladować funkcjonalność wyrażeń regularnych Perla i jest używany przez wiele nowoczesnych narzędzi.

Elementy wyrażeń regularnych

- Każdy znak, oprócz znaków specjalnych, określa sam siebie, np. `a` określa łańcuch złożony ze znaku a.
- Kolejne symbole oznaczają, że w łańcuchu muszą wystąpić dokładnie te symbole w dokładnie takiej samej kolejności, np. `ab` oznacza że łańcuch musi składać się z litery a poprzedzającej literę b.
- Kropka **.** oznacza dowolny znak z wyjątkiem znaku nowego wiersza (zależnie od ustawień i rodzaju wyrażeń).
- Znaki specjalne poprzedzone odwrotnym ukośnikiem **** powodują, że poprzedzanym znakom nie są nadawane żadne dodatkowe znaczenia i oznaczają same siebie, np. `\.` oznacza znak kropki (a nie dowolny znak).

Elementy wyrażeń regularnych

- Zestaw znaków między nawiasami kwadratowymi **[]** oznacza jeden dowolny znak objęty nawiasami kwadratowymi, np. `[abc]` oznacza a, b albo c. Można używać także przedziałów: `[a-f]`. Między nawiasami kwadratowymi:
  - daszek **^** na początku zestawu oznacza wszystkie znaki oprócz tych z zestawu;
  - aby uniknąć niejasności, znaki **-** (łącznik) i **]** (zamknięcie nawiasu kwadratowego) zapisywane są na skraju zestawu lub po znaku odwrotnego ukośnika, daszek zaś wszędzie z wyjątkiem początku łańcucha;
  - większość znaków specjalnych w tym miejscu traci swoje znaczenie.
- Pomiędzy nawiasami okrągłymi **()** grupuje się symbole do ich późniejszego wykorzystania.

Elementy wyrażeń regularnych

- Gwiazdka ***** po symbolu (nawiasie, pojedynczym znaku) nazywana jest domknięciem Kleene'a i oznacza zero lub więcej wystąpień poprzedzającego wyrażenia, np. `[a-f]*` oznacza dowolną liczbę powtórzeń symboli do `a` do `f`.
- Znak zapytania **?** po symbolu oznacza najwyżej jedno (być może zero) wystąpienie poprzedzającego wyrażenia.
- Plus **+** po symbolu oznacza co najmniej jedno wystąpienie poprzedzającego go wyrażenia.

Elementy wyrażeń regularnych

- W nawiasach klamrowych **{ }** podajemy liczbę powtórzeń, np. jeśli napiszemy `a{3}` oznacza ty, że a ma być powtórzone trzykrotnie, `a` jeśli napiszemy `b{2,4}` oznacza, że `b` na być powtórzone od 2 do 4 razy.
- Daszek **^** oznacza początek wiersza, dolar **$** oznacza koniec wiersza.
- Pionowa kreska **|** to operator alternatywy np. jeśli napiszemy `a|b|c` oznacza to, że w danym wyrażeniu może wystąpić `a` lub `b` lub `c`.

Przykłady wyrażeń regularnych

- Polski kod pocztowy składa się z sekwencji dwóch cyfr, myślnika i trzech cyfr:
  `[0-9]{2}-[0-9]{3}`
- Napis reprezentujący liczbę rzeczywistą składa się z opcjonalnego znaku, przynajmniej jednej cyfry, opcjonalnej części ułamkowej, która składa się z kolei z kropki dziesiętnej i przynajmniej jednej cyfry:
  `[+-]?[0-9]+(\.[0-9]+)?`
- Adres poczty elektronicznej (w wersji uproszczonej):
  ```
  ^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*@
  [a-zA-Z0-9-]+(\.[a-zA-Z0-9-]{1,})*
  \.([a-zA-Z]{2,}){1}$
  ```

Elementy wyrażeń regularnych

- Najczęściej używane klasy znaków mają zdefiniowane nazwy:
  - `[:alnum:]` dowolny znak alfanumeryczny
  - `[:alpha:]` dowolny znak alfabetu
  - `[:blank:]` dowolny biały znak, który nie jest separatorem linii
  - `[:digit:]` dowolna cyfra dziesiętna
  - `[:graph:]` dowolny znak graficzny
  - `[:lower:]` dowolna mała litera
  - `[:print:]` dowolny drukowalny znak
  - `[:punct:]` dowolny znak interpunkcyjny
  - `[:space:]` dowolny biały znak
  - `[:upper:]` dowolna wielka litera

Elementy wyrażeń regularnych

- Niektóre klasy znaków można definiować przy użyciu uproszczonej notacji:
  - `\w` litera, cyfra lub znak podkreślenia [_[:alnum:]]
  - `\d` cyfra dziesiętna [[:digit:]]
  - `\l` mała litera [[:lower:]]
  - `\s` odstęp (spacja, tabulacja itp. [[:space:]]
  - `\u` wielka litera [[:upper:]]

Przykłady wyrażeń regularnych

- Identyfikatory w języku C++ (znak podkreślenia lub litera, po których może występować zero lub więcej liter, cyfr i znaków podkreślenia):
  `[_[:alpha:]]\w*`

Wyrażenia regularne

- Standardowe narzędzia do tworzenia i posługiwania się wyrażeniami regularnymi w C++ znajdują się w pliku nagłówkowym **<regex>**.
- Korzystając z wyrażeń regularnych, można wykonywać następujące operacje:
  - **dopasowywać** dane wejściowe do wyrażenia regularnego;
  - **wyszukiwać** wzorców, które pasują do wyrażenia regularnego;
  - **dzielić** łańcuch na podciągi zgodnie z separatorem ciągów określonym za pomocą wyrażenia regularnego;
  - **zastępować** ciągi w pierwszym lub kolejnych podciągach pasujących do wyrażenia regularnego.

Metody regex_match() i regex_search()

- **regex_match()** dopasowuje wyrażenie regularne do łańcucha (sprawdza, czy cała sekwencja znaków pasuje do wyrażenia regularnego); przykład:

```
bool found = regex_match(
    "Znacznik XML: <tag>value</tag>",
    regex(".*<(.*)>.*</\\1>.*")
); // pasuje
```

- **regex_search()** szuka łańcucha pasującego do wyrażenia regularnego w strumieniu danych (sprawdza, czy sekwencja znaków częściowo pasuje do wyrażenia regularnego); przykład:

```
bool found = regex_search(
    "Znacznik XML: <tag>value</tag>",
    regex(R"(<(.*)>.*</\1>)")
); // pasuje
```

Metody regex_match() i regex_search()

- Przykłady:
- ```
  regex reg1("<.*>.*</.*>");
  bool found = regex_match(
      "<tag>value</tag", reg1);
  ```
- ```
  regex reg2("<(.*)>.*</\\1>");
  found = regex_match("<tag>value</tag>", reg2);
  ```
- ```
  bool found = regex_search(
      "Znacznik XML: <tag>value</tag>",
      regex(R"(<(.*)>.*</\1>)")
  ); // pasuje
  ```
  ```

Obsługa podwyrażeń

- Obiekty **`match_results<>`** można przekazać do składowych regex_match() i regex_search() w celu uzyskania szczegółów dopasowania.
- Klasa std::match_results<> jest szablonem. Jego egzemplarz musi zostać stworzony przez typ iteratora przetwarzanych znaków.
- Biblioteka standardowa C++ dostarcza predefiniowanych egzemplarzy:
  - **`smatch`** w odniesieniu do szczegółów dopasowania łańcuchów string;
  - **`cmatch`** w odniesieniu do szczegółów dopasowania łańcuchów w stylu języka C (const char*);
  - `wsmatch` i `wcmatch` dla długich znaków.

Obsługa podwyrażeń

```
smatch m; // do zwracanych szczegółów dopasowania
String data; // …
bool found = regex_search(data, m, regex("<(.*)>(.*)</(\\1)>"));
// wyświetlenie szczegółów dopasowania:
cout << "m.empty(): " << boolalpha << m.empty() << endl;
cout << "m.size(): " << m.size() << endl;
if (found) {
    cout << "m.str(): " << m.str() << endl;
    cout << "m.length(): " << m.length() << endl;
    cout << "m.position(): " << m.position() << endl;
    cout << "m.prefix().str(): " << m.prefix().str() << endl;
    cout << "m.suffix().str(): " << m.suffix().str() << endl;
    cout << endl;
    …
```

Obsługa podwyrażeń

```
    // iterowanie po wszystkich dopasowaniach
    // z wykorzystaniem indeksu dopasowania:
    for (int i = 0; i < m.size(); ++i) {
        cout << "m[" << i << "].str(): " << m[i].str() << endl;
        cout << "m.str(" << i << "): " << m.str(i) << endl;
        cout << "m.position(" << i << "): " << m.position(i) <<
    endl;
    }
    cout << endl;
    // iterowanie po wszystkich dopasowaniach j.w.:
    cout << "dopasowania:" << endl;
    for (auto pos = m.begin(); pos != m.end(); ++pos) {
        cout << " " << *pos << " ";
        cout << "(długość: " << pos->length() << ")" << endl;
    }
}
```

Obsługa podwyrażeń

- Obiekt match_results zawiera:
  - obiekt sub_match — m[0] zawierający wszystkie dopasowane znaki;
  - prefix() — obiekt sub_match reprezentujący wszystkie znaki przed pierwszym znakiem dopasowania;
  - suffix() — obiekt sub_match reprezentujący wszystkie znaki za ostatnim znakiem dopasowania.
- Dodatkowo dla każdej grupy przechwytywania mamy dostęp do odpowiadającego jej obiektu sub_match — m[n]. Ponieważ w wyrażeniu regularnym <(.*)>(.*)</(\1)> zaprezentowanym w tym przykładzie występują trzy grupy przechwytywania — jedna opisuje znacznik otwierający, druga wartość i trzecia znacznik zamykający, to są one dostępne jako obiekty m[1], m[2] i m[3].
- Składowa size() zwraca liczbę obiektów sub_match (z m[0] włącznie).
- Wszystkie obiekty sub_match są pochodnymi klasy pair<>. Pozycja pierwszego znaku jest dostępna w składowej first, natomiast pozycja za ostatnim znakiem jest dostępna jako składowa second. Dodatkowo składowa str() zwraca znaki w postaci łańcucha, składowa length() zwraca liczbę znaków, operator << zapisuje znaki do strumienia. Dostępna jest również niejawna konwersja typu do łańcucha znaków.

Obsługa podwyrażeń

- Dodatkowo obiekt match_results jako całość zawiera:
  - składową funkcję str() zwracającą dopasowany ciąg znaków jako całość (wywołanie str() lub str(0)) albo jako n-ty dopasowany podciąg (wywołanie str(n)); łańcuch jest pusty, jeśli nie istnieje dopasowany podciąg (zatem przekazanie argumentu *n większego od size() jest prawidłowe);*
  - składową funkcję length() zwracającą długość dopasowanego ciągu znaków jako całości (wywołanie length() lub length(0)) albo długość n-tego dopasowanego podciągu (wywołanie length(n)); składowa zwraca 0, jeśli nie istnieje dopasowany podciąg (zatem przekazanie argumentu n większego od size() jest prawidłowe);
  - składową funkcję position() zwracającą pozycję dopasowanego ciągu jako całości (wywołanie position() lub position(0)) albo pozycję n-tego dopasowanego podciągu (wywołanie position(n));
  - składowe funkcje begin(), cbegin(), end() i cend() do iterowania po obiektach sub_match od m[0] do m[n].

Iteratory dopasowań

- Do iterowania po wszystkich dopasowaniach wyrażenia regularnego możemy wykorzystać specjalne iteratory – iteratory te są typu **regex_iterator<>** i są one wyspecjalizowane dla łańcuchów **sregex_iterator** i ciągów znaków **cregex_iterator** oraz dla długich znaków wsregex_iterator lub wcregex_iterator.

Iteratory dopasowań – przykład

```
string data = "<osoba>\n"
    "   <imie>Nico</imie>\n"
    "   <nazwisko>Jankowski</nazwisko>\n"
    "</osoba>\n";
regex reg("<(.*)>(.*)</(\\1)>");
// wykorzystanie iteratora regex_iterator
// w celu przetwarzania wszystkich dopasowanych
    sregex_iterator beg(data.cbegin(), data.cend(), reg);
    sregex_iterator end;
    for_each (beg, end,
        [] (const smatch &m) {
            cout << "dopasowanie: " << m.str() << endl;
            cout << "   znacznik: " << m.str(1) << endl;
            cout << "    wartość: " << m.str(2) << endl;
        }
    );
```

Zastępowanie wyrażeń regularnych

- Aby zastąpić sekwencje znaków, które pasują do wyrażenia regularnego, należy użyć funkcji **regex_replace().**
- W ostatnim parametrze, który jest zamiennikiem, możemy wykorzystać dopasowane podwyrażenia, używając znaku dolara:
  - $& – dopasowany wzorzec,
  - $1, $2, … – $n to n-ta dopasowana grupa przechwytywania (numerowanie zaczynamy od 1, bo $0 to całe dopasowanie),
  - $` – prefiks dopasowanego wzorca,
  - $’ – sufiks dopasowanego wzorca,
  - $$ – znak dolara.

Zastępowanie wyrażeń regularnych

```
string data = "<osoba>\n"
    " <imie>Norbert</imie>\n"
    " <nazwisko>Jankowski</nazwisko>\n"
    "</osoba>\n";
regex reg("<(.*)>(.*)</(\\1)>");
// wyświetlenie danych z zamiennikami
// dopasowanych wzorców
cout << regex_replace(data, // dane
    reg, // wyrażenie regularne
    "<$1 value=\"$2\" />") // zamiennik
    << endl;
```

Wyjątki w wyrażeniach regularnych

- Podczas parsowania wyrażeń regularnych może dojść do wielu nieprzewidzianych sytuacji – biblioteka standardowa C++ zapewnia specjalną klasę przeznaczoną do obsługi wyjątków związanych z wyrażeniami regularnymi: jest to pochodna klasy **std::regex_error**, która dostarcza dodatkową składową code() służącą do zwracania kodu błędu.
- Dzięki składowej code() można dowiedzieć się, jaki błąd wystąpił podczas przetwarzania wyrażeń regularnych.
- Niestety, kody błędów zwracane przez składową code() są specyficzne dla implementacji, dlatego nie można wyświetlić ich bezpośrednio.

Wyjątki w wyrażeniach regularnych

- W celu obsługi wyjątków związanych z wyrażeniami regularnymi należy skorzystać ze stałych zdefiniowanych w pliku nagłówkowym <regex>:
  - std::regex_constants::error_collate,
  - std::regex_constants::error_ctype,
  - std::regex_constants::error_escape,
  - std::regex_constants::error_backref,
  - std::regex_constants::error_brack,
  - std::regex_constants::error_paren,
  - std::regex_constants::error_brace,
  - std::regex_constants::error_badbrace,
  - std::regex_constants::error_range,
  - std::regex_constants::error_space,
  - std::regex_constants::error_badrepeat,
  - std::regex_constants::error_complexity,
  - std::regex_constants::error_stack.

Literatura

- N.M.Josuttis: C++. Biblioteka standardowa. Podręcznik programisty. Wydanie 2. Rozdział 14: wyrażenia regularne. Wydawnictwo Helion, Gliwice 2014.
- B.Stroustrup: Język C++. Kompendium wiedzy. Wydanie 4. Rozdział 37: wyrażenia regularne. Wydawnictwo Helion, Gliwice 2014.
- C++. Wyrażenia regularne. https://learntutorials.net/pl/cplusplus/topic/1681/wyrazenia-regularne
- Wyrażenia regularne (C++11 / boost). https://cpp0x.pl/artykuly/Inne-artykuly/C++-Wyrazenia-regularne-C++11-boost/47