Robert Nowak

Wyrażenia regularne w C++
biblioteka boost::regex i boost::xpressive

Wyrażenia regularne uważa się za elementy programowania deklaratywnego, ponieważ nie dostarcza się sposobu postępowania (algorytmu), tylko danych wejściowych i opisu wyniku. Są one przydatne przy obróbce danych tekstowych, pozwalają znacznie uprościć kod programu. W artykule omawiamy dwie biblioteki dostarczające wyrażeń regularnych dla C++ mając na uwadze fakt, że udogodnienia takie zostały dodane do standardu tego języka.

Język C++ pozwala na przetwarzanie danych tekstowych, biblioteka standardowa dostarcza klasę do reprezentacji napisu oraz wyrażenia regularne, są one częścią standardu C++11. Tradycyjne napisy przetwarza się za pomocą języków skryptowych (AWK, sed, Python, Perl), które stanowią rozszerzenie systemu operacyjnego i są używane do kontroli pracy aplikacji, co często wymaga obróbki tekstowych plików konfiguracyjnych i tekstowych plików z logami. Jeżeli wydajność modułu operującego na tekście i utworzonego w tych jezykach jest niewystarczająca, warto rozważać użycie C++ do jego implementacji, wtedy kod jest translowany do kodu maszynowego, natomiast języki skryptowe są interpretowane – więc zazwyczaj dużo mniej wydajne.

Napisy w C++ są obiektami standardowej klasy std::string. Klasa ta jest konkretyzacją szablonu basic_string dla typu char (znaki 8 bitowe). Klasa std::wstring jest generowana na podstawie tego samego szablonu – znaki są przechowywane przez obiekty typu wchar_t (znaki 16 bitowe). Oprócz możliwości przechowywania napisów (o zmiennej długości) szablon basic_string dostarcza podstawowych operacji: porównywania, łączenia napisów, wyszukiwania znaków.

Język C++ pozwala na zapis stałych napisowych bezpośrednio w kodzie programu. Ciąg znaków w cudzysłowach jest zamieniany na tablicę znaków, którą można inicjować obiekty typu string (w opisie będziemy pomijali przestrzeń nazw). Można wskazać sposób kodowania oraz reprezentację znaku:

- „ASCII” tablica obiektów typu char (8 bitowe);
- L”wchar_t” tablica obiektów wchar_t (16 bitowe);
- u8”UTF-8 String \u03CO” tablica obiektów typu char, kodowanie UTF-8, znak UTF 03C0 - to pi;
- u”UTF-16 String \u03C0” tablica obiektów char16_t (16 bitowe), kodowanie UTF-16;
- U”UTF-32 String \u03C0”. tablica obiektów char32_t (32 bitowe), kodowanie UTF-32.

Tablice znaków są zakończone znakiem końca napisu, tzn. znakiem \0, jest to standard stosowany w C. Tablicę znaków można utworzyć za pomocą metody c_str() dla obiektu std::string, można także użyć takiej tablicy do inicjacji obiektu std::string.

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do kompilatora C++ oraz edytora tekstu. Przykłady wykorzystują biblioteki boost (www.boost.org). Aby poprawnie uruchomić te, które wykorzystują bibliotekę boost::regex, należy dodać odpowiednie zależności dla konsolidatora (linkera), dla g++ należy dodać opcję: -lboost_regex; dla Visual Studio (program link) biblioteka ta jest dodawana automatycznie. Przykłady wykorzystujące boost::xpressive wymagają jedynie poprawnej instalacji tej biblioteki. Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła umieszczono jako materiały pomocnicze.

Wyrażenia regularne pozwalają opisywać grupę „podobnych” napisów. Opisy takie są przydatne, gdy chcemy wykonywać pewne operacje na napisach, które reprezentują pewien szablon. Wyrażenia regularne są przydatne przy przetwarzaniu tekstów, dlatego są dostępne standardowo w tych językach skryptowych. Zostały także dodane do biblioteki standardowej w nowej wersji standardu C++11, oferują ją m.in. kompilatory Visual Studio 2010 oraz GNU GCC 4.6. Dla innych, stosowanych obecnie kompilatorów, możemy wykorzystać jedną z kilku bibliotek boost.

Wyrażenia regularne można opisać napisem, który zawiera symbole specjalne, patrz ramka. Przykładowo: .a oznacza dwuliterowy napis kończący się literą a (1a, aa, Aa, ...); [AEO]la oznacza napis Ala, Ela lub Ola; \da albo [[:digit:]]a oznacza dwuliterowy napis rozpoczynający się cyfrą, a kończący literą a (1a, 2a, ... ,9a); a*b oznacza dowolną liczbę liter a, a następnie literę b (b, ab, aab, ...); a? b oznacza opcjonalne wystąpienie litery a, a następnie literę b, czyli opisuje dwa napisy: b i ab; (a|b)b oznacza wystąpienie litery a lub b na pierwszej pozycji, a następnie litery b; (ab)+ oznacza przynajmniej jedno powtórzenie ciągu ab (ab, abab, ababab, ...); a(a|b)*b oznacza wyrażenie regularne, które opisuje napisy rozpoczynające się literą a, zakończone literą b, składające się jedynie z liter a i b (napisy ab, aab, abb, aaab, aabb, abab, abbb, ...).

ZNAKI PISARSKIE są kodowane w sposób standardowy od XIX wieku, używano wtedy kodu Morsa oraz kodów telegraficznych (5 bitów na znak). W systemach komputerowych dominującą rolę odgrywa ASCII (American Standard Code for Information Interchange, 7 bitowy), a ostatnio Unicode (znaki 16- i 32 bitowe oraz o zmiennej długości od 8 do 32 bitów).

Taka różnorodność wynika z faktu istnienia wielu alfabetów narodowych i wielu metod ich obsługi. Jeżeli wykorzystujemy ASCII, 7 bitów jest wykorzystywane na kodowanie znaków alfabetu łacińskiego, używanych w tekstach utworzonych w języku angielskim, cyfr i znaków przestankowych. Do reprezentacji innych znaków, na przykład znaków narodowych wykorzystano ósmy bit, co daje możliwość zakodowania dodatkowych 128 symboli. Niestety, dodatkowych znaków, które chcielibyśmy zakodować jest znacznie więcej niż 128, więc istnieje wiele standardowych zbiorów konwertujących 8 bitów na znak pisarski, zbiór taki nazywamy stroną kodową. Reprezentacja 8 bitowa oraz mechanizm stron kodowych, pozwala reprezentować teksty w większości używanych języków. Niestety takie kodowanie nie pozwala, w prosty sposób, umieszczać w tym samym tekście znaków z różnych stron kodowych, więc jeżeli tekst zawiera napisy w dwóch różnych językach jest kłopotliwy do kodowania. Z drugiej strony, te same ciągi bajtów mogą reprezentować różne ciągi znaków pisarskich, co może prowadzić do trudności z odczytaniem tekstu.

Wymienione problemy rozwiązano, stosując dłuższe, niż 1-bajtowe, reprezentacje znaków. Znaki 16 bitowe (UTF16, UCS2) pozwalają zakodować symbole używane w większości współczesnych języków, znaki 32 bitowe (UTF32, UCS-4) pozwalają kodować wszystkie symbole, także występujące w alfabetach obecnie nie stosowanych (np. hieroglify). Wadą napisów wykorzystujących kodowanie szerokimi znakami, jest zajętość pamięci, gdy kodujemy napisy w języku angielskim. Wadę tę można eliminować, stosując kodowanie znaków przez obiekty o różnej długości: znaki ASCII przez symbole 1-bajtowe (8 bitów), znaki z alfabetów narodowych przez symbole dłuższe (2, 3 lub 4 bajty). Takie kodowanie jest opisane przez standard UTF8. Zaletą UTF8 jest zgodność z ASCII, wadą – niebanalne obliczanie długości napisu, nie ma prostej zależności liczby znaków w napisie od liczby bajtów zajmowanych przez napis.

Obecnie najbardziej popularne jest kodowanie UTF-8. Język C++ wspiera wszystkie wymienione wyżej standardy.

Za pomocą przedstawionego opisu można utworzyć obiekt reprezentujący wyrażenie regularne. Obiekt taki pozwala typowo na następujące operacje:

- badanie, czy dany napis jest opisywany przez wyrażenie regularne;
- badanie, czy dany napis zawiera ciąg symboli (pod-napis) opisywany przez wyrażenie regularne;
- zamiana ciągu symboli opisywanych wyrażeniem regularnym na inny ciąg symboli.

Dodatkowo dostępne są metody iteracji po odnalezionych pod-napisach opisywanych wyrażeniem regularnym oraz dzielenia tekstu na grupy.

W artykule będziemy wykorzystywali bibliotekę boost::regex, która jest dostępna od kilku lat dla wielu platform i jest w dużej mierze zgodna ze standardem. Dodatkowo opisana będzie biblioteka boost::xpressive, która pozwala tworzyć obiekty reprezentujące wyrażenia regularne w czasie kompilacji. Obiekty reprezentujące wyrażenie regularne przedstawione w tym artykule mają semantykę wartości, można je kopiować, przekazywać jako argument czy zwracać z funkcji lub metody.

BIBLIOTEKA BOOST::REGEXP

Wyrażenie regularne jest obiektem typu szablonowego basic_regex, gdzie parametrem jest typ znaku. Dla znaków char klasa ta nazywa się regex, dla wchar_t – wregex. Napis opisujący wyrażenie regularne jest przekazywany w konstruktorze, gdzie tworzony jest wewnętrzna reprezentacja wyrażenia regularnego (jest nią automat skończony). Jeżeli obiekt jest prawidłowo zainicjowany, co oznacza m.in., że opis jest poprawny, to metoda basic_regex::empty() zwraca false. Opis wyrażenia regularnego to napis zawierający symbole specjalne, tak jak pokazano wcześniej. Jeżeli symbol posiadający specjalne znaczenie chcemy wykorzystać wprost, stawiamy przed nim lewy ukośnik (backslash, \). Proszę zwrócić uwagę, że kasowanie specjalnego znaczenia symbolu w wyrażeniu regularnym odbywa się za pomocą tego samego znaku, co kasowanie specjalnego znaczenia wewnątrz stałych napisowych języka C++, więc jeżeli wyrażenie regularne

Wybrane na potrzeby demonstracji symbole specjalne stosowane przy opisie wyrażeń regularnych. Pełna lista (kilkadziesiąt symboli) jest dostępna w dokumentacji bibliotek.

Symbol	Opis
^ $	początek i koniec linii
.	dowolny pojedynczy znak
[aeo]	zbiór znaków
[a-z]	zakres znaków
[^0-9]	znak spoza zakresu
[[:alpha:]]	litera (predefiniowany zbiór znaków)
[[:digit:]]	cyfra (predefiniowany zbiór znaków)
[[:space:]]	biały znak (predefiniowany zbiór znaków)
\d	to samo co [[:digit:]]
\s	to samo co [[:space:]]
*	dowolna liczba (zero lub więcej) powtórzeń poprzedniego symbolu
+	jedno lub więcej powtórzenie poprzedniego symbolu
?	opcjonalność: zero lub jedno powtórzenie
{m,n}	od m do n wystąpień
(wyr)	grupa, do której odnoszą się inne operacje
|	alternatywa

jest dostarczone jako stała napisowa (napis w cudzysłowach umieszczony w kodzie), to musimy backslash pisać podwójnie!

Po poprawnym utworzeniu obiektu reprezentującego dane wyrażenie regularne możemy używać funkcji opisanych poniżej. Badanie, czy napis jest opisywany wyrażeniem regularnym jest realizowane przez funkcję regex_match, wyszukiwanie ciągu znaków (pod-napisu) opisywanego danym wyrażeniem wykonuje regex_search, regex_replace pozwala zastępować fragmenty opisane danym wyrażeniem regularnym innym napisem.

Funkcja regex_match wymaga podania dwóch argumentów: napisu który będzie badany oraz obiektu reprezentującego wyrażenie regularne. Zwraca ona wartość logiczną, czy napis jest opisywany wyrażeniem regularnym. Przykład użycia tej funkcji zawiera Listing 1, gdzie pokazano funkcję correctIP sprawdzającą, czy napis jest poprawnym adresem IP. Przedstawiony kod sprowadza badanie tej poprawności do badania warunku, czy napis składa się z 4 sekcji zawierających od 1 do 3 cyfr i rozdzielonych kropką, nie jest to badanie doskonałe, za poprawny adres IP zostaną uznane napisy takie jak 256.0.0.0 czy 0.0.0.999. Wyrażenia regularne pozwalają taką funkcję zapisać za pomocą dwóch linii kodu.

Funkcja regex_search bada, czy dany napis zawiera ciąg symboli (pod-napis) opisywany przez wyrażenie regularne, zwraca ona prawdę logiczną (true), jeżeli taki ciąg udało się odnaleźć. Istnieje możliwość uzyskania wskazania na ten pasujący pod-napis, obiekt typu smatch, przekazywany jako argument referencyjny do funkcji regex_search jest wypełniany odpowiednimi wartościami. Przykład zawiera Listing 2, który pokazuje kod wyszukujący odpowiedni znacznik html w łańcuchu znaków.

```
bool correctIP(const std::string& ip) {
  static const regex
  e("\\d{1,3}\\.\\d{1,3}\\.\\
d{1,3}\\.\\d{1,3}");
  return regex_match(ip, e);
}
```

Listing 1. Badanie, czy napis opisuje poprawny adres IP

Obiekty generowane przez szablon match_result, zawierające kolekcję par wskaźników lub par iteratorów, są wykorzystywane do wskazywania na ciąg znaków. Pierwszy wskaźnik (lub pierwszy iterator) z pary wskazuje na pierwszy znak opisywanego ciągu, zaś drugi – na znak następny po ostatnim znaku opisywanego ciągu (stosuje się tutaj konwencję taką, jak dla zakresu iteratorów w bibliotece standardowej C++). Biblioteka definiuje cztery różne typy, generowane na podstawie tego szablonu (różne typy znaków, wskaźnik lub iterator):

- typedef match_results<const char*> cmatch;
- typedef match_results<const wchar_t*> wcmatch;
- typedef match_results<string::const_iterator> smatch;
- typedef match_results<wstring::const_iterator> wsmatch.

Pierwszy element kolekcji wskazuje na ciąg opisywany całym wyrażeniem regularnym, drugi element kolekcji (element o indeksie 1) opisuje ciąg „pasujący” do pierwszej grupy (grupy w wyrażeniu regularnym tworzone są za pomocą nawiasów), trzeci element kolekcji to ciąg odpowiadający drugiej grupie itd.

```
string html = /* wczytaj stronę html  */;
regex mail("href=\"mailto:(.*?)\">");
smatch what;//przechowuje wyniki wyszukiwania
if( regex_search(html,what,mail) ) {
   //what[0] napis pasujący do wyrażenia
   cout << "mail to" << what[1] << endl;
}
```

Listing 2. Wyszukiwanie wzorca opisywanego wyrażeniem regularnym. Tutaj kod, który przeszukuje tekst html w poszukiwaniu adresów e-mail

W przykładzie pokazanym na Listingu 2 wykorzystujemy niezachłanne dopasowywanie do grupy, co jest zapisane w wyrażeniu regularnym jako (.*?). Symbole wieloznaczne, na przykład *, mogą reprezentować pod-napisy różnej długości, co pozwala na powstawanie przypadków, gdy ten sam napis wejściowy może być interpretowany na wiele sposobów. Biblioteka boost::regex wykorzystuje dwa rodzaje dopasowań: zachłanne, gdy symbol opisuje pod-napis o maksymalnej długości, oraz niezachłanne, gdy dopasowanie oznacza ciąg o minimalnej długości. W ten sposób usuwane są niejednoznaczności. Wyrażenie regularne <(.*)> wskaże na ciąg Hello dla wejścia xxx Hello yyy, zaś wyrażenie <(.*?)> znajdzie dla tego samego wejścia.

Funkcja regex_replace pozwala zastępować fragmenty opisane danym wyrażeniem regularnym wybranym napisem. Argumentami tej funkcji są: napis wejściowy, wyrażenie regularne oraz napis formatujący, czyli napis, który będzie wstawiany w miejsce ciągu opisanego wyrażeniem regularnym. Napis formatujący może zawierać kilka symboli specjalnych, między innymi znak &, który określa fragment tekstu opisany wyrażeniem regularnym; znak \D oznacza fragment tekstu dopasowany do

Operatory reprezentujące niektóre symbole specjalne dla boost::xpressive

WYRAŻENIE	TYPOWY SYMBOL	OPIS
bos	^	początek sekwencji
a >> b	brak	konkatenacja
as_xpr('a')	a	znak
as_xpr("abc")	abc	ciąg znaków
(set='a','b','c')	[abc]	zbiór zbiór
_	.	dowolny znak
|	|	alternatywa
*	*	dowolna liczba wystąpień, - operator jest prefiksowy
+	+	jedno lub więcej wystąpienie
?	!	opcjonalność, prefiksowy
(s1= a), gdzie 1 to nr grupy	(a)	grupa

Więcej w książce

Omówienie współcześnie stosowanych technik, wzorce projektowe, programowanie generyczne, prawidłowe zarządzanie zasobami przy stosowaniu wyjątków, programowanie wielowątkowe, ilustrowane przykładami stosowanymi w bibliotece standardowej i bibliotekach boost, opisano w książce Robert Nowak, Andrzej Pająk ,,Język C++: mechanizmy, wzorce, biblioteki", BTC 2010.

grupy D, gdzie D = 1, 2, ..., 9. Przykładowo regex_replace("(<p>)+", "<p><p>hello", "<p>") zwróci napis <p>hello, zaś regex_replace("a,bx", reg, "\\2 = \\1") zwróci napis b = ax.

BIBLIOTEKA BOOST::XPRESSIVE - WYRAŻENIA W CZASIE KOMPILACJI

Biblioteka boost::xpressive jest inną biblioteką, wchodzącą w skład zbioru boost, która dostarcza obiektów i funkcji pozwalających wykorzystywać wyrażenia regularne. Oferuje ona udogodnienia takie jak opisane poprzednio, a dodatkowo pozwala tworzyć obiekty reprezentujące wyrażenie regularne w czasie kompilacji. Biblioteka boost::regex umożliwia jedynie zaszycie napisu opisującego wyrażenie w kodzie, automat skończony (reprezentacja wyrażenia regularnego) jest tworzony w konstruktorze tego obiektu - czyli w czasie działania. Tworzenie reprezentacji wyrażenia regularnego w czasie kompilacji pozwala zaoszczędzić czas podczas działania aplikacji, nie trzeba wtedy przetwarzać napisu opisującego wyrażenie i budować reprezentacji wewnętrznej.

Budowa obiektu reprezentującego wyrażenie regularne wykorzystuje techniki związane z meta-programowaniem, opisane m.in. w książce Nowak, Pająk „Język C++: mechanizmy, wzorce, biblioteki". Obiekt taki jest tworzony na podstawie wyrażenia, a nie na podstawie napisu. Wyrażenie to wykorzystuje obiekty klas dostarczanych przez bibliotekę, dla których przeciążono operatory reprezentujące symbole specjalne używane w wyrażeniach regularnych. Opis wyrażenia regularnego jest wyrażeniem w C++, co wymusza stosowanie innych, niż typowe, symboli w niektórych przypadkach. Podstawowe elementy takich wyrażeń zostały przedstawione w ramce, pozostałych kilkadziesiąt jest opisanych w dokumentacji biblioteki.

Przy tworzeniu opisu wyrażenia regularnego za pomocą wyrażeń C++ jesteśmy zmuszeni zapisywać symbol konkatenacji (łączenia kolejnych części opisujących wyrażenie regularne), czyli stosować operator >>. Opisując wyrażenie regularne napisem nie było takiej konieczności, napis ^a oznaczał wystąpienie litery a na początku linii, tutaj musimy to zapisać jako bos >> a, gdzie bos oznacza obiekt (stałą) reprezentujący początek napisu.

```
string subject(const string& in) {
  smatch w; //pod-napisy dopasowane do grup
  if( regex_match(in, w, bos>>"Subject: "
      >>(s1=*(as_xpr("Re: ")|as_xpr("Odp:
")))
      >>(s2=*_) ) ) {
    return string(w[2].first, w[2].second);
  }
  return "";
}
```

Listing 3. Funkcja dostarcza tytuł listu, pomijając przedrostki dodawane przez programy pocztowe

Znak lub ciąg znaków zapisujemy jako wynik działania funkcji as_xpr, operatory działają na obiektach pewnych typów, a nie na stałych znakowych i stałych napisowych. Jeżeli kompilator jest w stanie jednoznacznie zastosować konwersję, to możemy pominąć zapis as_xpr('a') i pisać 'a'. Dowolny znak nie jest oznaczany kropką, a podkreślnikiem, więc _ >> 'a' opisuje dwuliterowe napisy kończące się literą a; operator * i + jest prefiksowy, więc * as_xpr('a') >> 'b' oznacza dowolną liczbę liter a, a następnie literę b (b, ab, aab, ...); opcjonalne wystąpienie symbolu (lub grupy) zapisujemy jako ! , więc !as_xpr('a') >> 'b' oznacza opcjonalne wystąpienie litery a, a następnie literę b, czyli opisuje dwa napisy: b i ab. Wyrażenie bos >> "Subject: " >> (s1=*(as_xpr("Re: ")|as_xpr("Odp: "))) >> (s2=*_) wyszukuje w treści listu (e-mail) tytułu, pomijając przedrostki dodawane przez programy pocztowe. Identyczny obiekt można reprezentować napisem: Subject: (Re: |Odp:)*(.*). Wyrażenia takiego użyto na Listingu 3. Funkcja subject zwróci napis witajcie dla wejścia Subject: Re: Re: Odp: witajcie.

PODSUMOWANIE

Biblioteka boost::regex jest dostarczana jako biblioteka binarna, trzeba ją konsolidować. Biblioteka boost::xpressive zawiera tylko nagłówki, do jej użycia nie jest potrzebna konsolidacja. Wyrażenia regularne (takie jak boost::regex) są już dostępne dla popularnych kompilatorów, a będą powszechne, ponieważ są częścią biblioteki standardowej języka (standard C++11).

W Sieci

- http://www.boost.org – dokumentacja bibliotek boost,
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf – standard C++11 (working draft).

Robert Nowak rno@o2.pl

Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji wykorzystujących algorytmy sztucznej inteligencji i fuzji danych. Autor biblioteki faif.sourceforge.net. Programuje w C++ od ponad 15 lat.