

Przetwarzanie tekstu

„Nic, co jest oczywiste, nie jest takie oczywiste.
Użycie słowa »oczywisty« wskazuje na
brak logicznych argumentów.”

— Errol Morris

Ten rozdział został prawie w całości poświęcony wydobywaniu informacji z tekstu. Zapisujemy mnóstwo informacji w tej postaci, np. książki, listy e-mail, tabele itp. tylko po to, aby później je odzyskać w jakiejś łatwiejszej do przetwarzania przez komputer formie. Przejrzymy narzędzia biblioteki standardowej, które są najczęściej używane do przetwarzania tekstu — łańcuchy, strumienie wejścia i wyjścia oraz słowniki. Opiszemy podstawy wyrażeń regularnych, które są techniką pozwalającą definiować wzorce występujące w tekście. Na końcu pokażemy, jak za pomocą wyrażeń regularnych wydobywać z tekstu konkretne dane, jak kody pocztowe, oraz jak weryfikować format plików tekstowych.

23.1. Tekst

23.2. Łańcuchy

23.3. Strumienie wejścia i wyjścia

23.4. Słowniki

23.4.1. Szczegóły implementacyjne

23.5. Problem

23.6. Wyrażenia regularne

23.6.1. Surowe literały łańcuchowe

23.7. Wyszukiwanie przy użyciu wyrażeń regularnych

23.8. Składnia wyrażeń regularnych

23.8.1. Znaki i znaki specjalne

23.8.2. Rodzaje znaków

23.8.3. Powtórzenia

23.8.4. Grupowanie

23.8.5. Alternatywa

23.8.6. Zbiory i przedziały znaków

23.8.7. Błędy w wyrażeniach regularnych

23.9. Dopasowywanie przy użyciu wyrażeń regularnych

23.10. Źródła

23.1. Tekst

W zasadzie cały czas manipulujemy tekstem. Są nim wypełnione książki, jest nim większość tego, co widać na ekranie monitora, a także kod źródłowy programów. Wszelkiego rodzaju kanały komunikacji także roją się od słów. Wszystko, co jest przekazywane od człowieka do człowieka, można zaprezentować w postaci tekstu, ale lepiej nie zagłębiać się w to za bardzo. Obrazy i dźwięki zazwyczaj najlepiej prezentować w postaci graficznej i dźwiękowej (tzn., jako worki bitów). Wszystko pozostałe można pozostawić do programowej analizy tekstu i transformacji.

Strumieni wejścia i wyjścia oraz łańcuchów używamy od rozdziału 3., a więc tutaj tylko krótko przejrzymy te biblioteki. Słowniki (podrozdział 23.4) są szczególnie przydatne przy przetwarzaniu tekstu, dlatego przedstawimy przykład ich wykorzystania do analizowania tekstu wiadomości e-mail. Później skoncentrujemy się na wyszukiwaniu w tekście wzorców przy użyciu wyrażeń regularnych (podrozdziały 23.5 – 23.10).

23.2. Łańcuchy

Obiekt typu `string` składa się z sekwencji znaków oraz udostępnia kilka przydatnych operacji, np. dodawanie znaków, sprawdzanie długości łańcucha i łączenie łańcuchów. W istocie typ ten udostępnia całkiem sporo operacji, ale większość z nich jest przydatna tylko przy skomplikowanych działaniach na tekście na niskim poziomie. Opiszemy tylko niektóre bardziej użyteczne z nich. Szczegółowe informacje na ich temat (i pełną listę operacji) można znaleźć w podręczniku lub książce dla zaawansowanych. Można je znaleźć w nagłówku `<string>` (uwaga: nie w `<string.h>`):

Wybrane operacje na łańcuchach

<code>s1 = s2</code>	Przypisuje <code>s2</code> do <code>s1</code> . <code>s2</code> może być zwykłym łańcuchem lub łańcuchem w stylu języka C.
<code>s += x</code>	Dodaje <code>x</code> na końcu. <code>x</code> może być znakiem, zwykłym łańcuchem lub łańcuchem w stylu języka C.
<code>s[i]</code>	Indeksowanie.
<code>s1+s2</code>	Konkatenacja — znaki w powstałym łańcuchu będą kopią zawartości <code>s1</code> i <code>s2</code> .
<code>s1==s2</code>	Porównywanie łańcuchów. <code>s1</code> lub <code>s2</code> , ale nie oba na raz, może być łańcuchem w stylu języka C.
<code>s1<s2</code>	Porównywanie leksykograficzne łańcuchów. <code>s1</code> lub <code>s2</code> , ale nie oba na raz, może być łańcuchem w stylu języka C. Także <code><=</code> , <code>></code> oraz <code>>=</code> .
<code>s.size()</code>	Liczba znaków w <code>s</code> .
<code>s.length()</code>	Liczba znaków w <code>s</code> .
<code>s.c_str()</code>	Wersja w stylu języka C znaków w <code>s</code> .
<code>s.begin()</code>	Iterator wskazujący pierwszy znak.
<code>s.end()</code>	Iterator wskazujący ostatni znak.
<code>s.insert(pos,x)</code>	Wstawia <code>x</code> przed <code>s[pos]</code> . <code>x</code> może być zwykłym łańcuchem lub łańcuchem w stylu języka C. <code>s</code> rozszerza się, aby zrobić miejsce dla znaków z <code>x</code> .

Wybrane operacje na łańcuchach (ciąg dalszy)	
<code>s.append(x)</code>	Wstawia <code>x</code> za ostatnim znakiem <code>s</code> . <code>x</code> może być zwykłym łańcuchem lub łańcuchem w stylu języka C. <code>s</code> rozszerza się, aby zrobić miejsce dla znaków z <code>x</code> .
<code>s.erase(pos)</code>	Usuwa znaki z końca <code>s</code> , zaczynając od <code>s[pos]</code> . Po operacji rozmiar <code>s</code> wynosi <code>pos</code> .
<code>s.erase(pos,n)</code>	Usuwa <code>n</code> znaków z <code>s</code> , zaczynając od <code>s[pos]</code> . Po operacji rozmiar <code>s</code> wynosi <code>max(pos, size-n)</code> .
<code>pos = s.find(x)</code>	Znajduje <code>x</code> w <code>s</code> . <code>x</code> może być znakiem, zwykłym łańcuchem lub łańcuchem w stylu języka C. <code>pos</code> jest indeksem pierwszego znalezionej znaku lub <code>string::npos</code> (pozycja wskazująca koniec <code>s</code>).
<code>in>>s</code>	Wczytuje z <code>in</code> do <code>s</code> otoczone białymi znakami słowo.
<code>getline(in,s)</code>	Wczytuje wiersz tekstu z <code>in</code> do <code>s</code> .
<code>out<<s</code>	Zapisuje z <code>s</code> w <code>out</code> .

Operacje wejścia i wyjścia zostały opisane w rozdziałach 10. i 11., a ich zestawienie znajduje się w podrozdziale 23.3. Należy zauważyć, że operacje zapisu w łańcuchu powodują jego powiększenie, dzięki czemu nie może wystąpić przepełnienie.

Funkcje `insert()` i `append()` mogą przesuwać znaki, aby zrobić miejsce dla nowych znaków. Funkcja `erase()` przesuwa znaki w przód, aby zagwarantować, że po usunięciu znaku nie powstanie luka.

Typ łańcuchowy w bibliotece standardowej jest w istocie szablonem o nazwie `basic_string` obsługującym rozmaite zestawy znaków, np. Unicode, dając tym samym dostęp do tysięcy znaków (np. £, Ω, ∞, δ, ☉, poza „zwykłymi” znakami). Jeśli na przykład jest typ zawierający znak Unicode, jak Unicode, można napisać:

```
basic_string<Unicode> łańcuch_unicode;
```

Standardowy typ `string`, którego do tej pory używaliśmy, jest po prostu kontenerem `basic_string` znaków:

```
using string = basic_string<char> string; // string oznacza basic_string<char> (podrozdział 20.5)
```

Nie będziemy opisywać standardu Unicode ani łańcuchów Unicode, ale w razie potrzeby można znaleźć informacje na ich temat. Wówczas dowiesz się, że używa się ich (w języku, w typie `string`, w strumieniach wejścia i wyjścia i wyrażeniach regularnych) tak samo, jak zwykłych znaków. Jeśli musisz użyć znaków Unicode, najlepiej poradź się kogoś z większym doświadczeniem. Aby kod był użyteczny, musi nie tylko zostać napisany z zachowaniem reguł języka programowania, lecz również pewnych konwencji systemowych.

W kontekście przetwarzania tekstu należy zaznaczyć, że praktycznie wszystko można zaprezentować jako łańcuch znaków. Na przykład na tej stronie liczba 12.333 jest reprezentowana jako ciąg sześciu znaków (otoczonych spacjami). Jeśli ją wczytamy, przed jej użyciem w działaniach arytmetycznych musimy przekonwertować te znaki na liczbę zmiennoprzecinkową. W ten sposób pojawia się potrzeba konwersji wartości na łańcuchy i odwrotnie. W podrozdziale 11.4 pokazaliśmy, jak konwertuje się liczby całkowite na łańcuchy za pomocą strumienia `ostringstream`. Technikę tę można uogólnić, aby objęła wszystkie typy obsługujące operator `<<`:

```
template<typename T> string to_string(const T& t)
{
    ostringstream os;
    os << t;
    return os.str();
}
```

Na przykład:

```
string s1 = to_string(12.333);
string s2 = to_string(1+5*6-99/7);
```

Teraz `s1` ma wartość "12.333", a `s2` "17". W istocie funkcję `to_string()` można stosować także na rzecz innych typów niż liczbowe — każdej klasy `T`, która ma operator `<<`.

Konwersja w przeciwną stronę (typu `string` na wartości liczbowe) jest tak samo łatwa i przydatna:

```
struct bad_from_string : std::bad_cast
    // Klasa do raportowania błędów związanych z rzutowaniem łańcuchów
{
    const char* what() const override
    {
        return "Niepoprawne rzutowanie typu string.";
    }
};

template<typename T> T from_string(const string& s)
{
    istream is{s};
    T t;
    if (!(is >> t)) throw bad_from_string{};
    return t;
}
```

Na przykład:

```
double d = from_string<double>("12.333");

void do_something(const string& s)
try
{
    int i = from_string<int>(s);
    // ...
}
catch (bad_from_string e) {
    error ("Niepoprawny łańcuch na wejściu.",s);
}
```

Funkcja `from_string()` jest bardziej skomplikowana od `to_string()`, ponieważ typ `string` może reprezentować wartości różnych typów. Oznacza to, że trzeba poinformować, jakiego rodzaju wartość ma zostać wydobyta z łańcucha. Oznacza to również tyle, że badany łańcuch może nie zawierać reprezentacji wartości spodziewanego typu. Na przykład:

```
int d = from_string<int>("Gdyby kózka nie skakała"); // Ojej!
```

Tutaj więc jest możliwość wystąpienia błędu, który zaprezentowaliśmy w postaci wyjątku typu `bad_from_string`. W podrozdziale 23.9 zademonstrujemy, jak funkcja `from_string()` (lub jej odpowiednik) jest niezbędna w poważnym przetwarzaniu tekstu, ponieważ musimy wydobywać wartości liczbowe z pól tekstowych. W punkcie 16.4.3 pokazaliśmy użycie podobnej funkcji `get_int()` w kodzie GUI.

Należy zwrócić uwagę na podobieństwa w działaniu funkcji `to_string()` i `from_string()`. Zasadniczo stanowią swoje wzajemne przeciwieństwa, tj. (pomijając szczegóły typu białe znaki, zaokrąglanie itp.), dla każdego „sensownego typu T” mamy:

```
s==to_string(from_string<T>(s)) // dla wszystkich s
```

i

```
t==from_string<T>(to_string(t)) // dla wszystkich t
```

Słowo „sensowny” w tym kontekście oznacza, że typ T powinien mieć domyślny konstruktor, operator `>>` i odpowiadający mu operator `<<`.

Należy też zauważyć, że w implementacjach funkcji `to_string()` i `from_string()` całą ciężką pracę wykonuje strumień `stringstream`. Dzięki temu spostrzeżeniu można zdefiniować ogólną operację konwersji między dwoma dowolnymi typami z odpowiadającymi sobie operatorami `<<i>>>`:

```
template<typename Target, typename Source>
Target to(Source arg)
{
    stringstream interpreter;
    Target result;

    if (!(interpreter << arg)                // Zapisuje arg do strumienia
        || !(interpreter >> result)         // Wczytuje result ze strumienia
        || !(interpreter >> std::ws).eof()) // Zostało coś w strumieniu?
        throw runtime_error{"Błąd to<>()."};

    return result;
}
```

Ciekawa i sprytna instrukcja `!(interpreter>>std::ws).eof()` wczytuje białe znaki, które mogły zostać w strumieniu `stringstream` po wyodrębnieniu wyniku. Białe znaki są dozwolone, ale nie powinno być na wejściu nic więcej. Można się o tym przekonać, sprawdzając, czy osiągnięto koniec pliku. Jeśli więc spróbujemy wczytać liczbę typu `int` z łańcucha, to zarówno `to<int>("123")`, jak i `to<int>("123 ")` przejdą, a `to<int>("123.5")` nie, ze względu na `.5` na końcu.

23.3. Strumienie wejścia i wyjścia

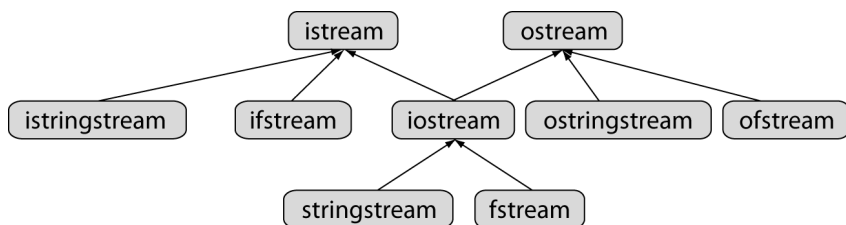


Rozmyślając o tym, co łączy łańcuchy z innymi typami, dochodzimy do strumieni wejścia i wyjścia. Biblioteka wejścia i wyjścia nie służy tylko do pobierania i wysyłania danych. Dodatkowo konwertuje formaty łańcuchowe na inne formaty i typy w pamięci. Strumienie wejścia i wyjścia biblioteki standardowej pozwalają na odczytywanie, zapisywanie i formatowanie łańcuchów znaków. Biblioteka `iostream` została opisana w rozdziałach 10. i 11., a więc tutaj przedstawimy tylko krótkie podsumowanie:

Strumień wejścia lub wyjścia

<code>in >> x</code>	Wczytuje z <code>in</code> do <code>x</code> zgodnie z typem <code>x</code> .
<code>out << x</code>	Zapisuje <code>x</code> w <code>out</code> zgodnie z typem <code>x</code> .
<code>in.get(c)</code>	Wczytuje znak z <code>in</code> do <code>c</code> .
<code>getline(in,s)</code>	Wczytuje wiersz z <code>in</code> do łańcucha <code>s</code> .

Strumienie standardowe są zorganizowane w hierarchię klas (podrozdział 14.3):



Klasy te pozwalają na zapisywanie i odczytywanie w plikach i łańcuchach (oraz wszystkim, co może je zastąpić, np. klawiaturze i ekranie — rozdział 10.). Jak napisaliśmy w rozdziałach 10. i 11., strumienie `iostream` udostępniają wyszukane narzędzia formatujące. Strzałki wskazują dziedziczenie (podrozdział 14.3), a więc obiektu klasy `stringstream` można użyć zamiast `iostream`, `istream` lub `ostream`.



Strumieni `iostream`, podobnie jak łańcuchów, można używać z większymi zestawami znaków, np. Unicode, w podobny sposób jak ze zwykłymi znakami. Pamiętaj, że jeśli planujesz używać znaków Unicode w operacjach wejścia i wyjścia, poproś o radę kogoś bardziej doświadczonego. Aby kod był użyteczny, musi spełniać nie tylko wymagania języka, lecz również odpowiadać konwencji systemowym.

23.4. Słowniki

Tablice asocjacyjne (słowniki, tablice skrótów) mają kluczowe znaczenie w wielu operacjach przetwarzania tekstu. Jest to związane z tym, że zbierając informacje, często kojarzy się je z łańcuchami tekstu, jak nazwy, adresy, kody pocztowe, numery ubezpieczenia, stanowiska pracy itp. Nawet mimo że niektóre z tych wartości można by było przekonwertować na liczby, często wygodniej i łatwiej traktować je jako identyfikujący tekst. Dobrym przykładem tego jest program liczący słowa z podrozdziału 21.6. Jeśli jeszcze nie opanowałeś słowników, przed dalszym czytaniem tego podrozdziału przeczytaj jeszcze raz podrozdział 21.6.

Weźmy jako przykład wiadomości e-mail. Często przeszukuje się i analizuje ich treść (i treść dzienników mailowych) i zazwyczaj wykorzystuje się do tego celu jakiś program (np. Thunderbird czy Outlook). Programy te w większości przypadków oszczędzają nam oglądania pełnego źródła wiadomości. Natomiast wszystkie informacje typu: nadawca, odbiorca, którędy wiadomość szła po wysłaniu i wiele innych, znajdują się w formacie tekstowym w nagłówku. Jest to kompletna wiadomość. Istnieją tysiące narzędzi do analizowania nagłówków. W większości z nich wykorzystywane są wyrażenia regularne (opisane w podrozdziałach 23.5 – 23.9), które pozwalają wydobywać informacje, oraz różne rodzaje tablic asocjacyjnych, które kojarzą ze sobą powiązane wiadomości. Często na przykład przeszukuje się pocztę, aby znaleźć wszystkie wiadomości wysłane przez jednego nadawcę, mające ten sam temat lub zawierające informacje na określony temat.

Posłużymy się bardzo uproszczonym plikiem poczty do zilustrowania niektórych technik wydobywania danych z plików tekstowych. Nagłówki te odpowiadają specyfikacji RFC2822 ze strony www.faqs.org/rfcs/rfc2822.html. Spójrzmy:

xxx
xxx

From: Jan Zebra <jzebra@machine.example>
To: Maria Kowalska <kowalska@example.net>
Subject: Powiedzieć dzień dobry
Date: Fri, 21 Nov 1997 09:55:06 -0600
Message-ID: <1234@local.machine.example>

Chciałbym powiedzieć dzień dobry.
A więc, „Dzień dobry”.

From: Jan Kwiatkowski <jkwiatkowski@example.com>
To: Maria Kowalska <@machine.tld:kowalska@example.net>, , jzebra@test.example
Date: Tue, 1 Jul 2003 10:52:37 +0200
Message-ID: <5678.21-Nov-1997@example.com>

Witam wszystkich.

To: "Maria Kowalska: Personal Account" <kowalska@home.example>
From: Jan Zebra <jzebra@machine.example>
Subject: Re: Powiedzieć dzień dobry
Date: Fri, 21 Nov 1997 11:00:00 -0600
Message-ID: <abcd.1234@local.machine.tld>
In-Reply-To: <3456@example.net>
References: <1234@local.machine.example> <3456@example.net>

To jest odpowiedź na Twoją odpowiedź.

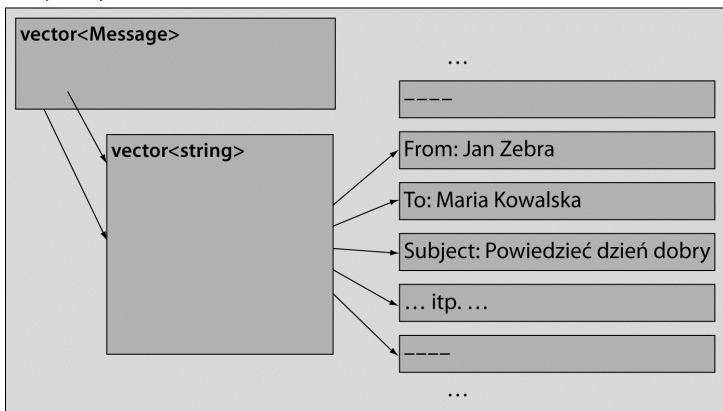
W istocie skróciliśmy plik, usuwając z niego większość informacji, oraz ułatwiliśmy jego przetwarzanie, kończąc każdą wiadomość wierszem zawierającym cztery myślniki. Napiszemy mały program znajdujący wszystkie wiadomości wysłane przez Jana Zebra i drukujący na wyjściu ich tematy (zawartość pola Subject). Jeśli nauczymy się to robić, będziemy umieli robić wiele ciekawych rzeczy.



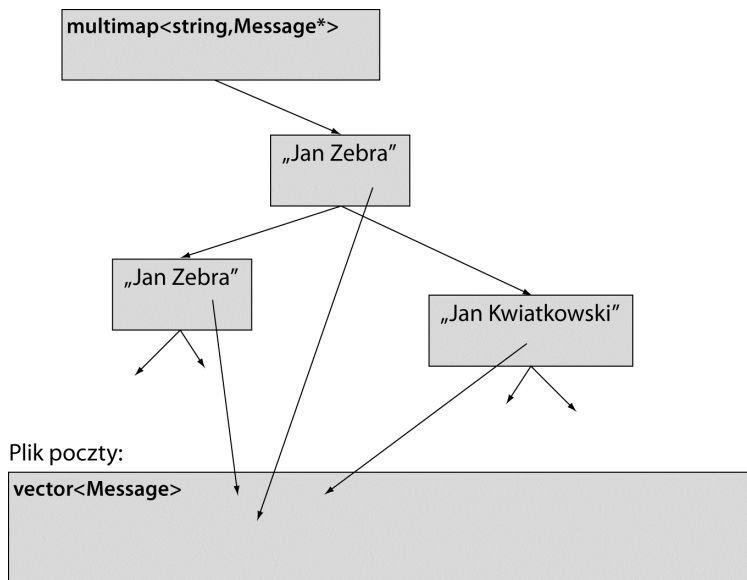
Najpierw trzeba zdecydować, czy dostęp do danych ma być swobodny czy będą analizowane podczas ich przepływu ze strumienia wejściowego. Wybierzemy pierwszą opcję, ponieważ w realnym programie często wyszukuje się kilku nadawców lub kilka informacji od jednego nadawcy. Poza tym to zadanie jest trudniejsze, dzięki czemu będziemy mogli na jego przykładzie opisać więcej technik. W szczególności ponownie będziemy korzystać z iteratorów.

Wczytamy cały plik poczty do struktury (którą nazwiemy `Mail_file`). Będzie ona przechowywała wszystkie wiersze tekstu tego pliku (w wektorze `vector<string>`) oraz znaczniki informujące, gdzie się kończy i zaczyna każda wiadomość (w wektorze `vector<Message>`).

Plik poczty:



Do tego dodamy iteratory i funkcje `begin()` i `end()`, aby móc przechodzić przez wiersze i wiadomości w normalny sposób. Dzięki temu będziemy mieli wygodny dostęp do wiadomości. Na tej podstawie napiszemy prosty program, który będzie gromadził wszystkie wiadomości od każdego nadawcy, aby mieć do nich wszystkich łatwy dostęp:



Na koniec wydrukujemy na wyjściu tematy wiadomości od Jana Zebry, aby zademonstrować sposób wykorzystania utworzonych wyżej struktur dostępowych.

Użyjemy wielu narzędzi z biblioteki standardowej:

```
#include<string>
#include<vector>
#include<map>
#include<fstream>
#include<iostream>
using namespace std;
```

Klasę Message zdefiniujemy jako parę iteratorów wektora `vector<string>` (naszego wektora przechowującego wiersze tekstu):

```
typedef vector<string>::const_iterator Line_iter;
class Message { // Message wskazuje pierwszy i ostatni wiersz wiadomości
    Line_iter first;
    Line_iter last;
public:
    Message(Line_iter p1, Line_iter p2) :first(p1), last(p2) { }
    Line_iter begin() const { return first; }
    Line_iter end() const { return last; }
    // ...
};
```

Mail_file będzie strukturą przechowującą wiersze tekstu i wiadomości:

```
using Mess_iter = vector<Message>::const_iterator;

struct Mail_file {
    // Mail_file przechowuje wszystkie wiersze z pliku
    // i ułatwia dostęp do wiadomości
    string name; // nazwa pliku
    vector<string> lines; // uporządkowane wiersze
    vector<Message> m; // uporządkowane wiadomości

    Mail_file(const string& n); // Wczytuje plik n do lines

    Mess_iter begin() const { return m.begin(); }
    Mess_iter end() const { return m.end(); }
};
```

Zwróć uwagę na dodanie iteratorów do struktur danych, aby ułatwić systematyczne ich przemierzanie. Nie będziemy tym razem używać algorytmów z biblioteki standardowej, ale gdybyśmy chcieli, to mamy już gotowe iteratory.

Aby znaleźć informację w wiadomości i ją wydobyć, potrzebne są dwie funkcje pomocnicze:

```
// Znajduje nazwę nadawcy wiadomości
// Zwraca true, jeśli znajdzie nazwę i
// zapisuje ją w s:
bool find_from_addr(const Message* m, string& s);

// Zwraca temat wiadomości, jeśli jest, lub "" w przeciwnym przypadku:
string find_subject(const Message* m);
```

Na koniec można napisać odpowiedni kod do wydobywania informacji z pliku:

```
int main()
{
    Mail_file mfile("my-mail-file.txt"); // Inicjalizacja mfile zawartością pliku

    // Najpierw gromadzone są wszystkie wiadomości w kontenerze multimap:

    multimap<string, const Message*> sender;

    for (const auto& m : mfile) {
        string s;
        if (find_from_addr(&m,s))
            sender.insert(make_pair(s,&m));
    }

    // Iteracja przez kontener multimap
    // i wydobywanie tematów wiadomości od Jana Zebry:
    auto pp = sender.equal_range("Jan Zebra <jzebra@machine.example>");
    for(auto p = pp.first; p!=pp.second; ++p)
        cout << find_subject(p->second) << '\n';
    }
```



Przyjrzymy się dokładnie sposobowi użycia słowników. Użyliśmy słownika multimap (podrozdział 20.10 i punkt B.4), ponieważ chcieliśmy zebrać w jednym miejscu wiadomości przychodzące spod tego samego adresu. Do tego doskonale nadaje się kontener multimap (ułatwia dostęp do elementów o takich samych kluczach). Oczywiście, jak zwykle, zadanie można podzielić na dwie części:

- budowa słownika,
- korzystanie ze słownika.

Słownik utworzymy poprzez przejście przez wszystkie wiadomości i wstawienie ich do kontenera za pomocą funkcji insert():

```
for (const auto& m : mfile) {
    const Message& m = *p;
    string s;
    if (find_from_addr(&m,s))
        sender.insert(make_pair(s,&m));
}
```

W słowniku zapisywane są pary (klucz, wartość), które tworzy się za pomocą funkcji make_pair(). Nasza „własnej roboty” funkcja find_from_addr() znajduje nazwę nadawcy.

Czemu najpierw zapisaliśmy wiadomości w wektorze, a dopiero później utworzyliśmy kontener multimap? Czemu od razu nie umieściliśmy ich w słowniku? Powód jest prosty i związany z fundamentalnymi zasadami:

- Najpierw tworzymy ogólną strukturę, której można użyć w wielu zastosowaniach.
- Następnie wykorzystujemy ją do konkretnego celu.

W ten sposób powstaje zbiór mniej lub bardziej nadających się do wielokrotnego użytku komponentów. Gdybyśmy od razu w `Mail_file` utworzyli słownik, musielibyśmy go przeddefiniowywać przy każdym innym zadaniu. Nasz słownik `multimap` (nazwany `sender`) jest sortowany według pól `Address` wiadomości. W większości innych zastosowań taki porządek nie byłby przydatny — mogłyby być potrzebne pola `Return`, `Recipients`, `Copy-to`, `Subject`, znaczniki czasu itd.

Zastosowanie takiej etapowej (lub **warstwowej**, ponieważ poszczególne części czasami nazywane są warstwami) techniki budowy aplikacji pozwala radykalnie uprościć projekt, implementację, dokumentację oraz utrzymanie programu. Chodzi o to, aby każda część wykonywała tylko jedno zadanie i robiła to bezpośrednio. Z drugiej strony, aby zrobić wszystko na raz, trzeba wykazać się sprytem. Oczywiście nasz program wydobywający informacje z nagłówek wiadomości e-mail jest bardzo mały. Wartość oddzielenia od siebie niepowiązanych rzeczy, podział na moduły i stopniowe rozbudowywanie aplikacji zwiększa się wraz z rozmiarem programu.

Wydobywamy informacje, znajdując wszystkie elementy o kluczu "Jan Zebra" za pomocą funkcji `equal_range()` (punkt B.4.10). Następnie przechodzimy iteracyjnie przez wszystkie elementy sekwencji [pierwszy, drugi] zwróconej przez tę funkcję i wydobywamy temat za pomocą funkcji `find_subject()`:

```
auto pp = sender.equal_range("Jan Zebra <jzebra@machine.example>");

for (auto p = pp.first; p!=pp.second; ++p)
    cout << find_subject(p->second) << '\n';
```

Wynikiem iteracji przez elementy słownika jest sekwencja par (klucz, wartość). Jak zawsze, pierwszy element pary (tutaj klucz typu `string`) nazywa się `first`, a drugi (tutaj wartość typu `Message`) nazywa się `second` (podrozdział 21.6).

23.4.1. Szczegóły implementacyjne

Oczywiście musimy zaimplementować funkcje, których używamy. Kusiło nas, aby uratować jakieś drzewo, zlecając to zadanie jako pracę domową, ale ostatecznie zdecydowaliśmy, że jednak dokończymy ten przykład sami. Konstruktor `Mail_file` otwiera plik oraz tworzy wektory `lines` i `m`:

```
Mail_file::Mail_file(const string& n)
    // Otwiera plik o nazwie n
    // Wczytuje wiersze z n do wektora lines
    // Znajduje wiadomości w lines i zapisuje je w m
    // Dla uproszczenia zakładamy, że każdą wiadomość kończy wiersz "———" {
    ifstream in {n};                                // Otwiera plik
    if (!in) {
        cerr << "nieudane " << n << '\n';
        exit(1);                                     // Zamyka program
    }
    for (string s; getline(in,s); ) // Buduje wektor wierszy
        lines.push_back(s);
```

```

auto first = lines.begin();    // Buduje wektor wiadomości
for (auto p = lines.begin(); p!=lines.end(); ++p) {
    if (*p == "——") {        // koniec wiadomości
        m.push_back(Message(first,p));
        first = p+1;          // —— nie jest częścią wiadomości
    }
}
}

```

Obsługa błędów to podstawa. Gdybyśmy mieli udostępnić ten program znajomym, musielibyśmy lepiej się postarać.



WYPRÓBUJ

Naprawdę zachęcamy Cię do uruchomienia tego przykładu i przeanalizowania wyniku, aby go dobrze zrozumieć. Jak powinna wyglądać lepsza obsługa błędów? Zmodyfikuj konstruktor struktury `Mail_file`, aby obsługiwał potencjalne błędy formatowania związane z użyciem łańcucha `"----`".

Funkcje `find_from_addr()` i `find_subject()` pełnią rolę zastępników, dopóki nie znajdziemy lepszego sposobu na identyfikowanie informacji w pliku (przy użyciu wyrażeń regularnych — podrozdziały 23.6 – 23.10):

```

int is_prefix(const string& s, const string& p)
    // Czy p jest pierwszą częścią s?
{
    int n = p.size();
    if (string(s,0,n)==p) return n;
    return 0;
}
bool find_from_addr(const Message* m, string& s)
{
    for (const auto& x : m)
        if (int n = is_prefix(x,"Od: ")) {
            s = string(x,n);
            return true;
        }
    return false;
}
string find_subject(const Message* m)
{
    for (const auto& x : m)
        if (int n = is_prefix(x,"Temat: ")) return string(x,n);
    return "";
}

```

Zwróć uwagę na sposób, w jaki używamy podłańcuchów — funkcja `string(s,n)` tworzy łańcuch składający się z końcówki łańcucha `s`, zaczynając od `s[n]` (`s[n]..s[s.size()-1]`), podczas gdy `string(s,0,n)` tworzy łańcuch zawierający znaki z przedziału `s[0]..s[n-1]`. Ponieważ funkcje te w istocie tworzą nowe łańcuchy i kopiują znaki, należy ostrożnie ich używać w miejscach, w których wydajność ma duże znaczenie.



Dlaczego funkcje `find_from_addr()` i `find_subject()` tak bardzo się różnią? Jedna na przykład zwraca typ `bool`, a druga `string`. Funkcje te są różne, ponieważ:



- `find_from_addr()` odróżnia znalezienie wiersza adresu z pustym adresem ("") i nieznanie adresu. W pierwszym przypadku funkcja ta zwraca wartość `true` (co oznacza, że znalazła adres) i ustawia `s` na "" (ponieważ adres ten jest pusty). W drugim przypadku zwraca wartość `false` (ponieważ nie ma wiersza adresu).
- `find_subject()` zwraca "", jeśli temat był pusty lub nie było w ogóle wiersza tematu.

Czy rozróżnienie zastosowane w funkcji `find_from_addr()` jest przydatne i niezbędne? Naszym zdaniem jest przydatne i zdecydowanie należy mieć jego świadomość. Przydaje się ono przy każdym szukaniu informacji w pliku z danymi — czy znaleźliśmy szukane pole oraz czy znajdowało się w nim coś ciekawego? W realnym programie funkcja `find_subject()` byłaby także napisana w podobnym stylu, jak `find_from_addr()`, aby pozwolić użytkownikom na dokonanie tego rozróżnienia.

Program ten nie jest zoptymalizowany pod względem wydajności, ale powinien być wystarczająco szybki do większości zastosowań. W szczególności wczytuje plik tylko raz i nie przechowuje wielu kopii tekstu z tego pliku. Gdyby pliki były duże, może korzystnie byłoby zamienić kontener `multimap` na `unordered_multimap`, ale nigdy nie wiadomo, dopóki się nie sprawdzi.

Wprowadzenie do kontenerów asocjacyjnych (`map`, `multimap`, `set`, `unordered_map` oraz `unordered_multimap`) biblioteki standardowej znajduje się w podrozdziale 21.6.

23.5. Problem

Strumień wejścia i wyjścia oraz łańcuchy pomagają w odczytywaniu i zapisywaniu sekwencji znaków, przechowywaniu ich oraz wstępnym przetwarzaniu. Jednak często się zdarza, że wykonujący operacje na tekście musi wziąć pod uwagę kontekst łańcucha lub zaangażować wiele podobnych łańcuchów. Rozważymy banalny przykład. Weźmiemy wiadomość e-mail (sekwencję słów) i sprawdzimy, czy zawiera skrót nazwy stanu w USA oraz kod pocztowy (dwie litery i pięć cyfr):

```
for (string s; cin>>s; ) {
    if (s.size()==7
        && isalpha(s[0]) && isalpha(s[1])
        && isdigit(s[2]) && isdigit(s[3]) && isdigit(s[4])
        && isdigit(s[5]) && isdigit(s[6]))
        cout << "znaleziono " << s << '\n';
}
```

Tutaj wywołanie `isalpha(x)` zwróci wartość `true`, jeśli `x` jest literą, a `isdigit(x)` zwróci `true`, jeśli `x` jest cyfrą (podrozdział 11.6).

To (zbyt) proste rozwiązanie ma kilka wad:

- Jest rozwlekłe (cztery wiersze i osiem wywołań funkcji).
- Pomija (może celowo?) każdy kod pocztowy, który nie jest oddzielony od otoczenia białymi znakami (np. "TX77845", TX77845-1234 oraz ATX77845).
- Pomija (może celowo?) każdy kod pocztowy zawierający spację między literami a cyframi (np. TX 77845).
- Akceptuje (może celowo?) każdy kod pocztowy zawierający małe litery (np. tx77845).
- Jeśli zechcemy poszukać kodu pocztowego w innym formacie (np. CB30FD), będziemy musieli napisać całkiem nowy kod.

Musi być lepszy sposób! Zanim go ujawnimy, rozważymy, jakie problemy by nas dotknęły, gdybyśmy pozostali przy „starym dobrym rozwiązaniu” polegającym na pisaniu większej ilości kodu, aby obsłużyć więcej przypadków.

- Aby obsłużyć więcej formatów, musielibyśmy dodać instrukcje `if` lub `switch`.
- Gdybyśmy chcieli obsługiwać małe i wielkie litery, musielibyśmy je jawnie przekonwertować (zwykle na małe) lub dodać kolejną instrukcję `if`.
- Musimy w jakiś sposób (ale jak?) opisać kontekst tego, co chcemy znaleźć. Oznacza to, że musimy posługiwać się pojedynczymi znakami zamiast łańcuchami, a to z kolei oznacza utratę wielu zalet stosowania strumieni i `ostream` (punkt 7.8.2).

Możesz napisać taki kod, jeśli chcesz, chociaż oczywiste jest, że jeśli podążymy tym tropem, nie unikniemy bałaganu związanego z obsługą specjalnych przypadków za pomocą instrukcji `if`. Nawet w tak prostym przykładzie trzeba zapewnić obsługę wielu alternatyw (np. pięcio- i dziewięciocyfrowych amerykańskich kodów pocztowych). W wielu innych przypadkach trzeba radzić sobie z powtórzeniami (np. dowolna liczba cyfr, po której znajduje się wykrzyknik, jak 123! lub 123456!). Trzeba by też było poradzić sobie z przedrostkami i przyrostkami. Jak zauważyliśmy w podrozdziałach 11.1 i 11.2, ludzka inwencja w zakresie formatów wyjściowych nie jest ograniczona dążeniem programistów do regularności i prostoty. Wystarczy przypomnieć sobie, jak wiele istnieje różnych formatów zapisu dat:

```
2007-06-05
5 czerwca 2007
5 cze 2007
5-6-2007
5.06.2007
5/6/2007
5/6/07
...
```

W tym momencie (jeśli nie wcześniej) doświadczony programista powie: „musi być jakiś lepszy sposób (niż pisanie zwykłego kodu)!” i zaczyna go poszukiwać. Najprostsze i najpopularniejsze rozwiązanie tego rodzaju problemów nosi nazwę **wyrażeń regularnych** (ang. *regular expression*). Stanowią one szkielet wielu programów przetwarzających tekst, są podstawą uniksowego

polecenia `grep` (zobacz 8. zadanie pracy domowej) oraz integralną część języków programowania, których często używa się do wykonywania tego typu zadań (np. `AWK`, `PERL` i `PHP`).

Implementacja wyrażeń regularnych, których będziemy tu używać, wchodzi w skład biblioteki standardowej języka `C++`. Jest ona zgodna z wyrażeniami regularnymi języka `PERL`, a więc można znaleźć na ich temat wiele podręczników i kursów. Można na przykład zajrzeć do artykułu roboczego komisji standaryzacyjnej języka `C++` (szukaj w internecie frazy `WG21`), dokumentacji biblioteki `boost::regex` Johna Maddocka oraz większości kursów języka `PERL`. My opiszemy tylko podstawy wyrażeń regularnych oraz niektóre najbardziej podstawowe sposoby ich wykorzystania.

WYPRÓBUJ

W dwóch ostatnich akapitach bezmyślnie użyliśmy kilku nazw i akronimów, nie podając ich objaśnień. Znajdź w internecie bliższe informacje na ich temat.

23.6. Wyrażenia regularne

Podstawowa zasada działania wyrażeń regularnych polega na tym, że definiuje się wzorzec, którego można poszukać w tekście. Pomyślmy, jak zwięźle zdefiniować wzorzec opisujący prosty amerykański kod pocztowy, np. `TX77845`. Oto pierwsza próba:

```
wwddddd
```

Litera `w` oznacza dowolną literę, a `d` dowolną cyfrę. Użyliśmy znaku `w` (od ang. *word* — słowo), ponieważ `l` (litera lub ang. *letter*) zbyt często myli się z cyfrą `1`. Powyższy zapis wystarcza dla tego prostego przykładu, ale spróbujemy teraz kodu w formacie dziewięciocyfrowym (np. `TX77845-5629`). Może coś takiego:

```
wwddddd-dddd
```

Wygląda dobrze, ale dlaczego `d` oznacza dowolną cyfrę, a `-` oznacza dokładnie to, na co wygląda — „zwykły myślnik”? Musimy w jakiś sposób zaznaczyć, że `w` i `d` mają specjalne znaczenie — reprezentują rodzaje znaków, a nie same siebie (`w` oznacza „a lub b lub c lub...”, podczas gdy `d` oznacza „1 lub 2 lub 3 lub...”). To jest za mało wyraźne. W związku z tym litery oznaczające nazwy rodzajów znaków będziemy poprzedzać wstecznym ukośnikiem w taki sam sposób, jak zawsze oznacza się znaki specjalne w `C++` (np. `\n` w literale łańcuchowym oznacza przejście do nowego wiersza). W ten sposób uzyskaliśmy:

```
\w\w\d\d\d\d\d-\d\d\d\d\d
```

Ten kod jest brzydki, ale przynajmniej jednoznaczny — ukośniki wskazują, że „dzieje się coś niezwykłego”. Powtarzalność określonego rodzaju znaku zaznaczamy, powtarzając jego symbol. Metoda ta może być zbyt żmudna i na dodatek stwarza ryzyko popełnienia błędu. Szybko: czy naprawdę dobrze zdefiniowaliśmy, że przed myślnikiem ma być pięć cyfr, a za nim cztery? Tak, ale nigdzie nie napisaliśmy cyfry `4` lub `5`, a więc aby się upewnić, musimy policzyć. Aby oznaczyć powtarzalność, można za znakiem postawić odpowiednią liczbę. Na przykład:

```
\w2\d5-\d4
```



Potrzebny jest nam jednak sposób na zaznaczenie, że cyfry 2, 5 i 4 w tym wzorcu oznaczają liczbę powtórzeń, a nie są zwykłymi znakami alfanumerycznymi. Użyjemy do tego klamer:

$$\backslash w\{2\}\backslash d\{5\}-\backslash d\{4\}$$

Przez to znak { zyskał specjalne znaczenie, podobnie jak \ (wsteczny ukośnik). Nie da się tego uniknąć, ale nie jest to takie złe.

Na razie wszystko jest dobrze, ale musimy poradzić sobie jeszcze z dwoma nieprzyjemnymi szczegółami: cztery ostatnie cyfry kodu pocztowego nie są obowiązkowe. Musimy mieć możliwość zaznaczenia, że akceptujemy zarówno TX77845, jak i TX77845-5629. Można to wyrazić na dwa sposoby:

$$\backslash w\{2\}\backslash d\{5\} \text{ lub } \backslash w\{2\}\backslash d\{5\}-\backslash d\{4\}$$

oraz:

$$\backslash w\{2\}\backslash d\{5\} \text{ i ewentualnie } -\backslash d\{4\}$$


Mówiąc jasno i precyzyjnie, najpierw trzeba wyrazić grupowanie (lub podwzorzec), aby móc mówić o częściach $\backslash w\{2\}\backslash d\{5\}$ i $-\backslash d\{4\}$ wzorca $\backslash w\{2\}\backslash d\{5\}-\backslash d\{4\}$. Zgodnie z konwencją do wyrażania grupowania stosuje się okrągłe nawiasy:

$$(\backslash w\{2\}\backslash d\{5\})(-\backslash d\{4\})$$

Podzieliliśmy wzorec na dwie części, a więc musimy poinformować, co chcemy z nimi zrobić. Jak zwykle kosztem wprowadzenia nowego narzędzia jest pojawienie się nowego specjalnego znaku: (jest teraz znakiem specjalnym, tak jak \ i {. Zgodnie z konwencją znak | oznacza „lub” (alternatywa), a ? oznacza coś warunkowego (opcjonalnego). Można zatem napisać:

$$(\backslash w\{2\}\backslash d\{5\})|(\backslash w\{2\}\backslash d\{5\}-\backslash d\{4\})$$

oraz:

$$(\backslash w\{2\}\backslash d\{5\})(-\backslash d\{4\})?$$

Podobnie jak klamer do zapisu liczby powtórzeń, znaku ? użyliśmy w formie przyrostkowej. Na przykład $(-\backslash d\{4\})?$ oznacza „opcjonalnie $-\backslash d\{4\}$ ”, tzn. akceptujemy na końcu cztery cyfry, przed którymi znajduje się myślnik. W istocie nawiasy otaczające wzorec pięciocyfrowego kodu $\backslash w\{2\}\backslash d\{5\}$ nie są do niczego potrzebne, a więc można napisać taki kod:

$$\backslash w\{2\}\backslash d\{5\}(-\backslash d\{4\})?$$

Aby uzupełnić rozwiązanie problemu postawionego w podrozdziale 23.5, możemy dodać opcjonalną spację za dwiema literami:

$$\backslash w\{2\} \text{ } ?\backslash d\{5\}(-\backslash d\{4\})?$$

Ten łańcuch " ?" wygląda nieco dziwnie — jest to spacja, po której znajduje się znak ?, co oznacza, że spacja jest opcjonalna. Aby ta spacja nie wyglądała, jakby był tam jakiś błąd, można ją umieścić w nawiasach:

$$\backslash w\{2\} (\text{ }) ?\backslash d\{5\}(-\backslash d\{4\})?$$

Gdyby ten sposób zapisu nadal był niejasny, można by było opracować notację do oznaczania białych znaków, np. `\s` (od słowa spacja). Wówczas można by było napisać taki kod:

```
\w{2}\s?\d{5}(-\d{4})?
```

Co by się stało, gdyby za literami znalazły się dwie spacje? Zgodnie z aktualną definicją wzorzec zaakceptowałby kody TX77845 i TX 77845, ale nie TX 77845. To jest nieco mało oczywiste. Musimy mieć możliwość zdefiniowania „zera lub więcej białych znaków”. W związku z tym wprowadzamy przyrostek `*` oznaczający „zero lub więcej”:

```
\w{2}\s*\d{5}(-\d{4})?
```

Ten kod jest zrozumiały dla tych, którzy przeczytali opis dochodzenia do niego. Ten sposób zapisu wzorców jest bardzo logiczny i nadzwyczaj zwięzły. Nie wybieraliśmy poszczególnych symboli losowo — przedstawiona przez nas notacja jest bardzo popularna i przydatna. Tego rodzaju kod trzeba pisać i czytać przy wielu zadaniach związanych z przetwarzaniem tekstu. Rzeczywiście, wygląda to, jakby kot przebiegł się po klawiaturze i jeden znak (nawet spacja) może całkowicie zmienić znaczenie kodu, ale trzeba po prostu się do tego przyzwyczaić. Nie znamy nic dużo lepszego, a poza tym notacja ta jest niezwykle popularna już od 30 lat, czyli od momentu pierwszego jej użycia w uniksowym poleceniu `grep` — a nawet wtedy nie była absolutną nowością.



23.6.1. Surowe literały łańcuchowe

Zwróć uwagę na ukośniki wsteczne w wyrażeniach regularnych. Aby użyć takiego ukośnika (`\`) w literale łańcuchowym C++, należy poprzedzić go dodatkowym ukośnikiem wstecznym. Spójrz na nasz wzorec kodu pocztowego:

```
\w{2}\s*\d{5}(-\d{4})?
```

Jeśli chcemy go przedstawić w postaci literału łańcuchowego, musimy napisać to tak:

```
"\\w{2}\\s*\\d{5}(-\\d{4})?"
```

Ponadto w wielu wzorcach może występować też cudzysłów prosty (`"`). Aby dodać go do literału łańcuchowego, przed nim także należy postawić ukośnik wsteczny. W ten sposób szybko wszystko może się poplątać. Problem z używaniem znaków specjalnych we wzorcach wyrażen regularnych jest na tyle uciążliwy, że w C++ i innych językach programowania wprowadzono pojęcie surowych literałów łańcuchowych, które ułatwiają pracę z tymi wzorcami. W takim surowym łańcuchu ukośnik wsteczny jest zwykłym znakiem (a nie znakiem specjalnym), podobnie jak cudzysłów prosty (który nie oznacza w tym przypadku końca łańcucha). W formie surowego literału łańcuchowego nasz wzorec kodu pocztowego wygląda tak:

```
R"(\w{2}\s*\d{5}(-\d{4})?)"
```

Człon `R"` (oznacza początek łańcucha, a `"` — koniec. Zatem właściwy łańcuch ma, nie licząc zera końcowego, 22 znaki:

```
\w{2}\s*\d{5}(-\d{4})?
```

23.7. Wyszukiwanie przy użyciu wyrażeń regularnych

Wykorzystamy utworzony w poprzednim podrozdziale wzorec kodu pocztowego do wyszukiwania kodów w pliku. Nasz program będzie definiował wzorec i wczytywał plik wiersz po wierszu, szukając odpowiadających mu ciągów. Jeśli znajdzie wystąpienie tekstu pasującego do wzorca, wydrukuje numer wiersza, w którym je znalazł oraz to, co znalazł:

```
#include <regex>
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int main()
{
    ifstream in {"file.txt"};           // plik wejściowy
    if (!in) cerr << "Brak pliku.\n";

    regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"}; // wzorec kodu pocztowego
    int lineno = 0;
    for (string line; getline(in, line); ) { // wczytuje wiersz do bufora
        ++lineno;
        smatch matches;                  // Tu trafiają pasujące łańcuchy
        if (regex_search(line, matches, pat))
            cout << lineno << ": " << matches[0] << '\n';
    }
}
```

Kod ten wymaga szczegółowego objaśnienia. Narzędzia obsługujące wyrażenia regularne biblioteki standardowej znajdują się w nagłówku `<regex>`. Programista może zatem zdefiniować następujący wzorec `pat`:

```
regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"}; // wzorec kodu pocztowego
```



Wzorec typu `regex` jest w istocie rodzajem łańcucha, więc można go zainicjalizować łańcuchem — w tym przypadku użyliśmy surowego literału łańcuchowego. Jednak `regex` to nie zwykły łańcuch. Skomplikowany mechanizm dopasowywania wzorców, który jest tworzony w chwili inicjalizacji obiektu typu `regex` (lub przypisania do niego), jest ukryty i jego opis wykracza poza zakres tematyczny tej książki. Wystarczy wiedzieć, że po zainicjowaniu obiektu `regex` naszym wzorcem kodu pocztowego można go zastosować wobec każdego wiersza pliku:

```
smatch matches;
if (regex_search(line, matches, pat))
    cout << lineno << ": " << matches[0] << '\n';
```



Funkcja `regex_search(line, matches, pat)` przeszukuje `line` w celu znalezienia wszystkiego, co pasuje do przechowywanego w `pat` wyrażenia regularnego. Jeśli coś znajdzie, zapisuje to w zmiennej `matches`. Jeśli funkcja ta nic nie znajdzie, zwraca wartość `false`.

Zmienna `matches` jest typu `smatch`. Litera `s` w nazwie tego typu oznacza „sub” czyli „pod” lub `string`. Zasadniczo `smatch` jest wektorem „poddopasowań” typu `string`. Pierwszy element (tutaj `matches[0]`) jest kompletnym dopasowanym łańcuchem. `matches[i]` można traktować jako łańcuch, jeśli `i < matches.size()`. Jeśli więc dla danego wyrażenia regularnego maksymalna liczba podwzorców wynosi `N`, to `matches.size() == N+1`.

Czym więc jest podwzorec? Pierwsza nasuwająca się dobra odpowiedź to: „Wszystko, co znajduje się w nawiasach we wzorcu”. We wzorcu `\w{2}\s*\d{5}(-\d{4})?` w nawiasach znajduje się czterocyfrowe rozszerzenie kodu pocztowego. Jest to jedyny podwzorec, który tam widzimy, a więc możemy zgadnąć, że `matches.size() == 2`. Podejrzewamy też, że można z łatwością uzyskać dostęp do tych końcowych czterech cyfr. Na przykład:



```
for (string line; getline(in,line); ) {
    smatch matches;
    if (regex_search(line, matches, pat)) {
        cout << lineno << ": " << matches[0] << '\n'; // całe dopasowanie
        if (1 < matches.size() && matches[1].matched)
            cout << "\t: " << matches[1] << '\n'; // poddopasowanie
    }
}
```

Mówiąc ściśle, nie musieliśmy pisać testu `1 < matches.size()`, ponieważ już dobrze przyjrzeliliśmy się wzorcowi. Jednak czuliśmy się jak paranoicy (ponieważ eksperymentowaliśmy na wielu wzorcach w `pat`, z których nie wszystkie miały tylko jeden podwzorec). Można spytać, czy poddopasowanie powiodło się, sprawdzając jego składową `matched` — tutaj `matches[1].matched`. Jeśli Cię to ciekawi, gdy `matches[i].matched` ma wartość `false`, niedopasowany podwzorec `matches[i]` jest drukowany jako pusty łańcuch. Analogicznie podwzorec, który nie istnieje, jak `matches[17]` dla powyższego wzorca, jest traktowany jako niedopasowany podwzorec.

Wypróbowaliśmy ten program na pliku zawierającym następującą treść:

```
address TX77845
ffff tx 77843 asasasaa
ggg TX3456-23456
howdy
zzz TX23456-3456sss ggg TX33456-1234
cvzcv TX77845-1234 sdsas
xxxTx77845xxx
TX12345-123456
```

Uzyskaliśmy następujący wynik:

```
wzorec: \w{2}\s*\d{5}(-\d{4})?
1: TX77845
2: tx 77843
5: TX23456-3456
: -3456
6: TX77845-1234
: -1234
7: Tx77845
8: TX12345-1234
: -1234
```

Należy zauważyć, że nasz program:

- Nie dał się oszukać źle sformatowanym kodom pocztowym w wierszu zaczynającym się od ggg (co tam jest źle?).
- Znalazł tylko pierwszy kod pocztowy w wierszu zaczynającym się od zzz (prosiłiśmy tylko o jeden z każdego wiersza).
- Znalazł poprawne przyrostki w wierszach 5. i 6.
- Znalazł kod pocztowy ukryty między ciągami xxx w wierszu 7.
- Znalazł (niestety?) kod pocztowy ukryty w TX12345-123456.

23.8. Składnia wyrażeń regularnych

Przedstawiliśmy podstawowy przykład wykorzystania wyrażeń regularnych. Nadszedł czas, aby potraktować je (w takiej formie, w jakiej są używane w bibliotece `regex`) trochę systematyczniej i pełniej.



Wyrażenia regularne (ang. *regular expressions*, *regexprs* lub *regexs*) to niewielki język służący do definiowania wzorców znakowych. Język ten jest ekspresywny i bardzo zwięzły, a przez to jego kod bywa tajemniczy. Po kilku dziesięcioleciach jego używania opracowano wiele własności i kilka dialektów. My opiszemy duży i przydatny jego podzbiór, który wydaje się, że jest najpowszechniej stosowany (dialekt języka PERL). Jeśli potrzebujesz więcej informacji, aby coś wyrazić lub zrozumieć kod kogoś innego, poszukaj ich w internecie. Istnieje mnóstwo kursów (różnej jakości) i specyfikacji.



Biblioteka ta obsługuje także notacje w stylu ECMAScript, POSIX, awk, grep oraz egrep, a także masę opcji przeszukiwania. To może być bardzo przydatne, zwłaszcza gdy trzeba dopasowywać wzorce napisane w innym języku. Jeśli chcesz poznać bardziej zaawansowane narzędzia niż opisane tutaj, możesz poszukać informacji na ich temat. Pamiętaj tylko, że używanie jak największej liczby narzędzi nie robi z Ciebie dobrego programisty. Jeśli to tylko możliwe, dbaj o biednego programistę od utrzymania kodu (możesz sam nim zostać za kilka miesięcy), który musi czytać i rozumieć Twój kod — pisz kod, który nie jest niepotrzebnie sprytny i w miarę możliwości unikaj mało znanych narzędzi.

23.8.1. Znaki i znaki specjalne

Za pomocą wyrażeń regularnych można definiować wzorce do dopasowywania znaków w łańcuchach. Standardowo znak we wzorcu pasuje do siebie samego w łańcuchu. Na przykład wyrażenie regularne "abc" zostanie dopasowane do ciągu abc w zdaniu Czy tu jest abc?.

Prawdziwa siła wyrażeń regularnych tkwi w ich znakach specjalnych oraz kombinacjach znaków, które mają specjalne znaczenie we wzorcach:

Znaki o specjalnym znaczeniu

- | | |
|---|---|
| . | dowolny pojedynczy znak (symbol wieloznaczny) |
| [| klasa znakowa |
| { | licznik |
-

Znaki o specjalnym znaczeniu	
(początek grupy
)	koniec grupy
\	następny znak ma specjalne znaczenie
*	zero lub więcej
+	jeden lub więcej
?	opcjonalność (zero lub jeden)
	alternatywa (lub)
^	początek wiersza; negacja
\$	koniec wiersza

Na przykład:

`x.y`

pasuje do każdego trzyznakowego łańcucha zaczynającego się od `x` i kończącego na `y`, np. `xy`, `x3y` czy `xay`, ale nie `xyx`, `3xy` czy `xy`.

Należy zauważyć, że `{...}`, `*`, `+` oraz `?` to operatory przyrostkowe. Na przykład `\d+` oznacza „jedna lub więcej cyfr dziesiętnych”.

Aby użyć specjalnego znaku we wzorcu, trzeba zastosować sekwencję specjalną ze znakiem wstecznego ukośnika. Na przykład znak `+` we wzorcu jest operatorem „jeden lub więcej”, ale zapis `\+` oznacza znak plusa.

23.8.2. Rodzaje znaków

Najczęściej spotykane kombinacje znaków są reprezentowane w zwięzłej postaci jako „znaki specjalne”:

Specjalne znaki dla rodzajów znaków		
<code>\d</code>	cyfra dziesiętna	<code>[[[:digit:]]</code>
<code>\l</code>	mała litera	<code>[[[:lower:]]</code>
<code>\s</code>	biały znak (spacja, tabulator itd.)	<code>[[[:space:]]</code>
<code>\u</code>	wielka litera	<code>[[[:upper:]]</code>
<code>\w</code>	litera (<code>a – z</code> lub <code>A – Z</code>) lub cyfra (<code>0 – 9</code>) lub znak podkreślenia (<code>_</code>)	<code>[[[:alnum:]]</code>
<code>\D</code>	nie <code>\d</code>	<code>[^[:digit:]]</code>
<code>\L</code>	nie <code>\l</code>	<code>[^[:lower:]]</code>
<code>\S</code>	nie <code>\s</code>	<code>[^[:space:]]</code>
<code>\U</code>	nie <code>\u</code>	<code>[^[:upper:]]</code>
<code>\W</code>	nie <code>\w</code>	<code>[^[:alnum:]]</code>

Należy zauważyć, że specjalny znak w postaci wielkiej litery stanowi negację znaku specjalnego w postaci takiej samej małej litery. Na przykład `\W` oznacza „nie litera”, a nie wielką literę.

W trzeciej kolumnie przedstawiono alternatywną składnię wykorzystującą dłuższe nazwy (np. `[[digit:]]`).

Podobnie jak `string` i `iostream`, biblioteka `regex` obsługuje duże zbiory znaków, takie jak `Unicode`. Podobnie jak w poprzednich przypadkach wspominamy tylko o tym, a znalezienie konkretnych informacji pozostawiamy czytelnikowi. Praca z tekstem zawierającym znaki z zestawu `Unicode` leży poza zakresem tematycznym tej książki.

23.8.3. Powtórzenia

Powtórzenia wzorców definiuje się za pomocą operatorów przyrostkowych:

Powtórzenia	
<code>{n}</code>	dokładnie <i>n</i> razy
<code>{n,}</code>	<i>n</i> lub więcej razy
<code>{n,m}</code>	przynajmniej <i>n</i> i najwyżej <i>m</i> razy
<code>*</code>	zero lub więcej, tj. <code>{0,}</code>
<code>+</code>	jeden lub więcej, tj. <code>{1,}</code>
<code>?</code>	opcjonalny (zero lub jeden), tj. <code>{0,1}</code>

Na przykład:

`Ax*`

pasuje do litery *A*, po której jest zero lub więcej liter *x*, np.

`A`
`Ax`
`Axx`
`AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`

Jeśli musi być przynajmniej jedno wystąpienie *x*, należy zamiast operatora `*` użyć `+`. Na przykład:

`Ax+`

pasuje do litery *A*, po której jest jedna lub więcej liter *x*, np.:

`Ax`
`Axx`
`AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`

ale nie:

`A`

Częsty przypadek zera lub jednego wystąpienia (opcjonalny) reprezentuje znak zapytania. Na przykład:

`\d-?\d`

pasuje do dwóch cyfr opcjonalnie rozdzielonych myślnikiem:

`1-2`
`12`

ale nie:

1--2

Do określania konkretnej liczby wystąpień lub konkretnego przedziału wystąpień należy używać klamer. Na przykład:

$\backslash w\{2\}-\backslash d\{4,5\}$

pasuje do dwóch liter i myślnika (-), po których znajdują się cztery lub pięć cyfr:

Ab-1234

XX-54321

22-54321

ale nie:

Ab-123

?b-1234

Tak, cyfry są znakami $\backslash w$.

23.8.4. Grupowanie

Aby wyznaczyć wyrażenie regularne jako podwzorzec, należy użyć operatora grupowania, którym są okrągłe nawiasy. Na przykład:

$(\backslash d^* :)$

Powyższy kod definiuje podwzorzec zera lub więcej cyfr, po których znajduje się dwukropek. Grupy można użyć jako części bardziej skomplikowanego wzorca. Na przykład:

$(\backslash d^* :)?(\backslash d^+)$

Powyższy kod definiuje opcjonalną i możliwe że pustą sekwencję cyfr, po której znajduje się dwukropek i jedna lub więcej cyfr. Nic dziwnego, że opracowano zwięzły i precyzyjny sposób wyrażania tego typu rzeczy!

23.8.5. Alternatywa

Znak „lub” (|) określa alternatywę. Na przykład:

Subject: $(FW:|Re:)?(.*)$

Ten wzorzec rozpoznaje wiersz tematu wiadomości e-mail z opcjonalnym polem FW: lub Re:, po którym znajduje się zero lub więcej znaków. Na przykład:

Subject: FW: Witaj, świecie!

Subject: Re:

Subject: Niebieskie norweskie

ale nie:

SUBJECT: Re: papugi

Subject FW: Brak tematu!

Pusta alternatywa jest niedozwolona:

```
(|def) // błąd
```

Można natomiast określić kilka alternatyw na raz:

```
(bs|Bs|bS|BS)
```

23.8.6. Zbiory i przedziały znaków

Specjalne znaki stanowią skrócony zapis reprezentujący najczęściej używane rodzaje znaków — cyfry (`\d`), litery, cyfry i podkreślenie (`\w`) itd. (punkt 23.7.2). Można łatwo i z dużym pożytkiem zdefiniować własne takie znaki. Na przykład:

<code>[\w @]</code>	litera, spacja lub znak @
<code>[a-z]</code>	małe litery od a do z
<code>[a-zA-Z]</code>	małe lub wielkie litery od a do z
<code>[Pp]</code>	mała lub wielka litera P
<code>[\w\~]</code>	litera lub myślnik
<code>[asdfghjkl;']</code>	znaki znajdujące się w środkowym rzędzie na standardowej klawiaturze
<code>[.]</code>	kropka
<code>[.{(\ *+?^\$}]</code>	znak o specjalnym znaczeniu w wyrażeniu regularnym

W specyfikacji rodzaju znaków myślnik oznacza przedział, np. `[1-3]` (1, 2 lub 3) lub `[w-z]` (w, x, y lub z). Należy ostrożnie korzystać z takich przedziałów, ponieważ różne języki mogą mieć różne litery alfabetu i nie każde kodowanie liter ma taką samą kolejność. Jeśli potrzebujesz przedziału niebędącego częścią najpowszechniejszego zestawu liter alfabetu angielskiego, zajrzyj do dokumentacji.

Należy zauważyć, że znaków specjalnych, jak `\w` (oznaczający „dowolny znak słowa”), można używać w specyfikacjach rodzajów znaków. Jak zatem wcielić wsteczny ukośnik (`\`) do klasy znaków? Jak zwykle posilujemy się znakiem specjalnym: `\\`.



Jeśli pierwszym znakiem specyfikacji rodzaju znaków jest `^`, oznacza on negację. Na przykład:

<code>[^aeiouy]</code>	żadna z tych samogłosek
<code>[^\\d]</code>	nie cyfra
<code>[^aeiouy]</code>	spacja, jedna z tych samogłosek lub ^

W ostatnim wyrażeniu regularnym `^` nie jest pierwszym znakiem za `[`, a więc występuje tam jako zwykły znak, a nie operator negacji. Wyrażenia regularne czasami wymagają bardzo dużej precyzji.

W implementacji regex znajduje się też zestaw nazwanych klas znaków. Jeśli na przykład trzeba dopasować dowolny znak alfanumeryczny (tj. literę lub cyfrę: a-z lub A-Z albo 0-9), można wykorzystać wyrażenie regularne `[[:alnum:]]`. Słowo `alnum` jest nazwą zestawu znaków (alfanumerycznych). Wzorzec niepustego łańcucha ujętych w cudzysłowy znaków alfanumerycznych mógłby być następujący: `"[[:alnum:]]+"`. Aby to wyrażenie regularne wstawić do zwykłego łańcucha, trzeba pozbyć się cudzysłowów za pomocą znaków specjalnych:

```
string s {"\" [[:alnum:]]+\""};
```


Ponadto, aby wstawić ten literał łańcuchowy do obiektu typu `regex`, trzeba w podobny sposób poradzić sobie z wstecznymi ukośnikami i cudzysłowami oraz zastosować inicjalizację przy użyciu operatora `()`, ponieważ konstruktor `regex` z łańcucha jest jawny:

```
regex s {"\\\\" [[:alnum:]]+\\\\""};
```

Prostszy kod uzyskujemy, używając literału łańcuchowego:

```
regex s2 {R"(" [[:alnum:]]+"")"};
```

Wzorce zawierające ukośniki wsteczne i cudzysłowy proste lepiej jest definiować w formie surowych literałów łańcuchowych. W wielu przypadkach oznacza to, że tak należy definiować większość wzorców.

Stosowanie wyrażeń regularnych wymusza trzymanie się wielu konwencji związanych z notacją. Poniżej przedstawiamy listę standardowych klas znaków:

Klasy znaków	
<code>alnum</code>	dowolny znak alfanumeryczny
<code>alpha</code>	dowolny znak alfabetu
<code>blank</code>	każdy biały znak, który nie jest oddzielnikiem linii
<code>cntrl</code>	dowolny znak sterujący
<code>d</code>	dowolna cyfra dziesiętna
<code>digit</code>	dowolna cyfra dziesiętna
<code>graph</code>	dowolny znak graficzny
<code>lower</code>	dowolny mały znak
<code>print</code>	dowolny drukowalny znak
<code>punct</code>	dowolny znak interpunkcyjny
<code>s</code>	dowolny biały znak
<code>space</code>	dowolny biały znak
<code>upper</code>	dowolny wielki znak
<code>w</code>	dowolny znak słowa (znaki alfanumeryczne plus znak podkreślenia)
<code>xdigit</code>	dowolny znak mogący stanowić cyfrę szesnastkową

Niektóre implementacje klasy `regex` mogą udostępniać większą liczbę klas znaków. Jednak przed użyciem takich nazw, których nie wymieniono w powyższej tabeli, należy się upewnić, że są wystarczająco przenośne.

23.8.7. Błędy w wyrażeniach regularnych

Co się stanie, jeśli w wyrażeniu regularnym pojawi się błąd? Rozważmy:

```
regex pat1 {"(|ghi)"}; // brakuje alternatywy
regex pat2 {"[c-a]"}; // nie przedział
```



Gdy wzorec zostaje przypisany do obiektu typu `regex`, jest on sprawdzany. Jeśli program dopasowujący wyrażenia regularne nie może go użyć ze względu na błędy lub zbyt skomplikowaną składnię, zostaje zgłoszony wyjątek `bad_expression`.

Poniżej przedstawiamy krótki program, który pozwala zorientować się, jak przebiega dopasowywanie wzorców wyrażen regularnych:

```
#include <regex>
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
using namespace std;

// Wzorec i zestaw wierszy zostają przyjęte na wejściu,
// Wzorec zostaje sprawdzony i następuje wyszukanie odpowiadających mu wierszy

int main()
{
    regex pattern;

    string pat;
    cout << "Wprowadź wzorec: ";
    getline(cin, pat); // Wczytuje wzorec

    try {
        pattern = pat; // Sprawdza pat
        cout << "Wzorec: " << pat << '\n';
    }
    catch (bad_expression) {
        cout << pat << " nie jest poprawnym wyrażeniem regularnym.\n";
        exit(1);
    }

    cout << "Wprowadź tekst:\n";
    int lineno = 0;

    for (string line; getline(cin, line); ) {
        ++lineno;
        smatch matches;
        if (regex_search(line, matches, pattern)) {
            cout << "Wiersz " << lineno << ": " << line << '\n';
            for (int i = 0; i < matches.size(); ++i)
                cout << "\tpasuje.[" << i << "]: "
                    << matches[i] << '\n';
        }
        else
            cout << "nie pasuje.\n";
    }
}
```

WYPRÓBUJ



Uruchom powyższy program i wypróbuj go na kilku wzorcach, np. `abc`, `x.*x`, `(.*)`, `\([^\)]*\)` oraz `\w+ \w+(Jr\.)?`.

23.9. Dopasowywanie przy użyciu wyrażeń regularnych

Wyrażenia regularne znajdują zastosowanie głównie w dwóch sytuacjach:



- **Szukanie** w (dowolnie długim) strumieniu danych łańcucha odpowiadającego wyrażeniu regularnemu — funkcja `regex_search()` szuka swojego wzorca jako podłańcucha w strumieniu.
- **Dopasowywanie** wyrażenia regularnego względem łańcucha (znanego rozmiaru) — funkcja `regex_match()` szuka pełnego dopasowania swojego wzorca i łańcucha.

Przykładem wyszukiwania było opisane w podrozdziale 23.6 wyrażenie regularne definiujące kody pocztowe. W tym podrozdziale pokażemy przykład dopasowywania. Weźmiemy pod uwagę wydobywanie danych z poniższej tabeli:

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER IALT
0A	12	11	23
1A	7	8	15
1B	4	11	15
2A	10	13	23
3A	10	12	22
4A	7	7	14
4B	10	5	15
5A	19	8	27
6A	10	9	19
6B	9	10	19
7A	7	19	26
7G	3	5	8
7I	7	3	10
8A	10	16	26
9A	12	15	27
0M0	3	2	5
0P1	1	1	2
0P2	0	5	5
10B	4	4	8
10CE	0	1	1
1M0	8	5	13
2CE	8	5	13
3DCE	3	3	6
4M0	4	1	5
6CE	3	4	7
8CE	4	4	8
9CE	4	9	13
REST	5	6	11
Alle klasser	184	202	386

Tabela ta (zawierająca dane z 2007 roku o liczbie uczniów w szkole podstawowej, do której chodził Bjarne Stroustrup) została wyodrębniona z kontekstu (strony internetowej), w którym wygląda bardzo ładnie. Jest to typowy przykład rodzaju danych do analizy:

- Zawiera pole z danymi liczbowymi.
- Zawiera pola z łańcuchami, których znaczenie jest jasne tylko dla tych, którzy znają kontekst tej tabeli (tutaj podkreśla to użycie języka duńskiego).
- Łańcuchy znaków zawierają spacje.
- Pola tych danych są oddzielone „znakiem oddzielającym”, którym w tym przypadku jest tabulator.



Postanowiliśmy wykorzystać tę tabelę, aby przedstawić typowy, a przy okazji niezbyt trudny przykład. Należy jednak zwrócić uwagę na jeden szczegół — nie potrafimy sami rozróżnić spacji i tabulatorów — musimy zostawić rozwiązanie tego problemu kodowi.

Zademonstrujemy zastosowanie wyrażeń regularnych do:

- Sprawdzenia, czy powyższa tabela jest poprawnie skonstruowana (tzn. każdy wiersz zawiera odpowiednią liczbę pól).
- Sprawdzenia, czy suma liczb się zgadza (w ostatnim wierszu powinna znajdować się suma wszystkich wierszy poszczególnych kolumn).



Jeśli potrafimy to zrobić, będziemy umieli wszystko! Możemy na przykład utworzyć nową tabelę, w której wiersze zawierające taką samą pierwszą cyfrę (oznaczającą rok — pierwsza klasa ma numer 1) są scalone, lub sprawdzić, czy w określonym przedziale lat liczba uczniów rośnie czy maleje (zobacz 10. i 11. zadanie pracy domowej).

Do przeanalizowania tej tabeli potrzebne będą dwa wzorce — jeden dla wiersza nagłówkowego i jeden dla pozostałych wierszy.

```
regex header {R"^(\\w ]+(          [\\w ]+)*$)";
regex row {R"^(\\w ]+(          \\d+)(  \\d+)(    \\d+)$)";
```



Pamiętaj, że pochwaliliśmy składnię wyrażeń regularnych za zwięzłość i użyteczność. Nie chwaliliśmy jej jednak za łatwość do zrozumienia dla początkujących. W istocie mają one zasłużoną reputację języka „tylko do pisania”. Zaczniemy od nagłówka. Ponieważ nie zawiera żadnych liczb, moglibyśmy go po prostu wyrzucić, ale dla zdobycia większej praktyki przeanalizujemy go. Wiersz ten zawiera cztery „pola ze słowami” (pola alfanumeryczne), które są oddzielone tabulatorami. Pola te mogą zawierać spacje, a więc nie możemy użyć po prostu `\\w`. Zamiast tego użyjemy wzorca `[\\w]`, który oznacza znak słowa (litera, cyfra lub znak podkreślenia) lub spację. Aby zaznaczyć, że może być jeden lub więcej takich znaków, piszemy `[\\w]+`. Pierwszy z nich powinien znajdować się na początku linii, a więc wzorec zamienia się w `^(\\w]+.` Daszek (znak `^`) oznacza „początek linii”. Każde z pozostałych pól można zdefiniować jako tabulator, po którym znajdują się jakieś słowa: `([\\w]+)`. Może być ich dowolna liczba, po której powinien znajdować się koniec linii: `([\\w]+)*$`. Symbol dolara oznacza „koniec linii”. Aby zapisać to jako surowy literał łańcuchowy w języku C++, trzeba dodać wsteczne ukośniki:

```
R"^(\\w ]+(          \\d+)(  \\d+)(    \\d+)$)"
```

Należy zauważyć, że nie da się wzrokowo rozpoznać tabulatorów, chociaż w tym przypadku rozszerzyły się w druku i dały o sobie znać.

Teraz czas przejść do ciekawszej części zadania, a więc wzorca dla wierszy, z których chcemy wydobyć dane liczbowe. Pierwsze pole jest takie samo, jak wcześniej — `^[\\w]+`. Za nim znajdują się dokładnie trzy pola liczbowe, z których każde jest poprzedzone tabulatorem: `(\\d+)`, a więc otrzymujemy:

```
^[\\w ]+(      \\d+)(      \\d+)(      \\d+)$
```

Po przerobieniu na surowy literał łańcuchowy:

```
R"^[\\w ]+(      \\d+)(      \\d+)(      \\d+)$"
```

Pozostaje tylko wykorzystać te wzorce. Najpierw sprawdzimy, czy tabela jest poprawnie skonstruowana:

```
int main()
{
    ifstream in("table.txt"); // plik wejściowy
    if (!in) error("Brak pliku wejściowego.\n");

    string line;                // bufor wejściowy
    int lineno = 0;

    regex header {R"^[\\w ]+(      [\\w ]+*$)"}; // wiersz nagłówkowy
    regex row {R"^[\\w ]+(      \\d+)(      \\d+)(      \\d+)$"}; // wiersz danych

    if (getline(in,line)) { // Sprawdza wiersz nagłówkowy
        smatch matches;
        if (!regex_match(line, matches, header))
            error("Brak nagłówka.");
    }
    while (getline(in,line)) { // Sprawdza wiersz danych
        ++lineno;
        smatch matches;
        if (!regex_match(line, matches, row))
            error("Niepoprawny wiersz.",to_string(lineno));
    }
}
```

Dla uproszczenia opuściliśmy dyrektywy `#include`. Ponieważ w każdym wierszu sprawdzamy wszystkie znaki, użyliśmy funkcji `regex_match()` zamiast `regex_search()`. Różnica między nimi polega na tym, że pierwsza musi dopasować wszystkie znaki pojawiające się na wejściu, a druga musi znaleźć tylko podłańcuch pasujący do jej wzorca. Jeśli ktoś przez przypadek wpisze `regex_match()` zamiast `regex_search()` (lub odwrotnie), może mieć duże problemy ze znalezieniem źródła błędów. Obie te funkcje jednak identycznie wykorzystują swój argument `matches`.

Teraz możemy przejść do weryfikacji danych w tabeli. Przechowujemy sumy uczniów w kolumnach chłopców („drenge”) i dziewcząt („piger”). Dla każdego wiersza sprawdzamy, czy ostatnie pole („ELEVER IALT”) rzeczywiście zawiera sumę dwóch poprzednich. Ostatni wiersz („Alle klasser”) ma ambicję stanowić sumę powyższych wierszy.

Aby to sprawdzić, musimy zmodyfikować wyrażenie `row`, aby uczynić pole tekstowe podpasowaniem, dzięki czemu możliwe będzie rozpoznanie „Alle klasser”:

```
int main()
{
    ifstream in{"table.txt"};                // plik wejściowy
    if (!in) error("Brak pliku wejściowego.");

    string line;                              // bufor wejściowy
    int lineno = 0;

    regex header {R"([^\w ]+(\w+)*$)"};        // wiersz nagłówka
    regex row {R"([^\w ]+(\d+)(\d+)(\d+)$)"}; // wiersz danych

    if (getline(in,line)) { // Sprawdza wiersz nagłówkowy
        smatch matches;
        if (regex_match(line, matches, header)) {
            error("Brak nagłówka.");
        }
    }

    // sumy kolumn:
    int boys = 0;
    int girls = 0;
    while (getline(in,line)) {
        ++lineno;
        smatch matches;
        if (!regex_match(line, matches, row))
            cerr << "Nieprawidłowy wiersz: " << lineno << '\n';

        if (in.eof()) cout << "na końcu pliku. \n";

        // Sprawdza wiersz:
        int curr_boy = from_string<int>(matches[2]);
        int curr_girl = from_string<int>(matches[3]);
        int curr_total = from_string<int>(matches[4]);
        if (curr_boy+curr_girl != curr_total) error("Błędna suma w wierszu \n");

        if (matches[1]=="Alle klasser") { // ostatni wiersz
            if (curr_boy != boys) error("Nie zgadza się suma chłopców.\n");
            if (curr_girl != girls) error("Nie zgadza się suma dziewcząt.\n");
            if (!(in>>ws).eof()) error("Za wierszem sumy są jakieś znaki.");
            return 0;
        }

        // Aktualizacja sum:
        boys += curr_boy;
        girls += curr_girl;
    }

    error("Nie znaleziono wiersza sum.");
}
```

Ostatni wiersz jest pod względem semantycznym inny niż pozostałe, ponieważ zawiera sumę ich wartości. Rozpoznajemy to po jego etykiecie („Alle klasser”). Zdecydowaliśmy się nie akceptować za nim więcej niebiałych znaków (przy użyciu techniki `to<>()`; — podrozdział 23.2) oraz zwrócić błąd, jeśli go nie znajdziemy.

Do wydobycia wartości całkowitoliczbowej z pól danych użyliśmy funkcji `from_string()` (podrozdział 23.2). Sprawdziliśmy już, czy te pola zawierają tylko cyfry, a więc nie musimy obawiać się o powodzenie konwersji `string` na `int`.

23.10. Źródła

Wyrażenia regularne to popularne i przydatne narzędzie. Są dostępne w wielu językach programowania i formatach. Wspiera je elegancka teoria oparta na językach formalnych oraz wydajna technika implementacji oparta na maszynach stanów. Pełny opis ogólności, teorii i implementacji wyrażań regularnych oraz stosowanie maszyn stanów nie mieszczą się w zakresie tematycznym tej książki. Jednak dzięki temu, że tematy te są praktycznie standardem w programach studiów informatycznych, oraz temu, iż są one bardzo popularne, można łatwo znaleźć na ich temat więcej informacji (jeśli Cię one interesują czy wręcz ich potrzebujesz):

Aho Alfred V., Lam Monica S., Sethi Ravi, Ullman Jeffrey D., *Compilers: Principles, Techniques, and Tools*, wydanie drugie (często nazywane „The Dragon Book”), Addison-Wesley, 2007.

Cox Russ, *Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, ...)*, <http://swtch.com/~rsc/regex/regex1.html>.

Maddock J., dokumentacja `boost::regex`, www.boost.org/doc/libs/1_39_0/libs/regex/doc/html/index.html.

Schwartz Randal L., Phoenix Tom, Foy Brian D., *Perl. Wprowadzenie. Wydanie czwarte*, Helion, 2006.

Ćwiczenia

1. Sprawdź, czy `regex` wchodzi w skład Twojej biblioteki standardowej. Wskazówka: wypróbuj `std::regex` i `tr1::regex`.
2. Uruchom program z podrozdziału 23.7. Może być konieczne znalezienie odpowiedniej konfiguracji opcji projektu i/lub wiersza poleceń, aby dołączyć bibliotekę `regex` i używać nagłówków `regex`.
3. Użyj programu z ćwiczenia 2. do przetestowania wzorców z podrozdziału 23.7.

Powtórzenie

1. Gdzie znajdziemy „tekst”?
2. Które narzędzia z biblioteki standardowej są najczęściej wykorzystywane do analizowania tekstu?
3. Czy funkcja `insert()` wstawia elementy przed czy za swoją pozycją (lub iteratorem)?
4. Co to jest `Unicode`?
5. Jak konwertuje się na reprezentację łańcuchową i odwrotnie (na inny typ i z innego typu)?

6. Jaka jest różnica między instrukcjami `cin>>s` i `getline(cin,s)` przy założeniu, że `s` to łańcuch?
7. Wymień standardowe strumienie.
8. Co to jest klucz w słowniku? Podaj przykłady przydatnych typów kluczy.
9. W jaki sposób przechodzi się przez elementy słownika?
10. Jaka jest różnica między kontenerem `map` a kontenerem `multimap`? Jakiej przydatnej operacji kontenera `map` nie ma w `multimap`? Dlaczego?
11. Jakie operacje są wymagane dla iteratora przechodzącego do przodu?
12. Jaka jest różnica między pustym a nieistniejącym polem? Podaj dwa przykłady.
13. Dlaczego do zapisania wyrażenia regularnego potrzebne są znaki specjalne?
14. Jak zamienia się wyrażenie regularne w zmienną typu `regex`?
15. Co oznacza wzorec `\w+\s\d{4}`? Podaj trzy przykłady. Jak wyglądałby literal łańcuchowy reprezentujący ten wzorec do zainicjowania zmiennej typu `regex`?
16. Jak w programie sprawdza się, czy łańcuch jest poprawnym wyrażeniem regularnym?
17. Co robi funkcja `regex_search()`?
18. Co robi funkcja `regex_match()`?
19. Jak reprezentuje się kropkę w wyrażeniach regularnych?
20. Jak wyrazić „przynajmniej trzy” w wyrażeniu regularnym?
21. Czy `7` jest znakiem typu `\w`? Czy jest nim znak podkreślenia (`_`)?
22. Jak wygląda notacja oznaczająca wielkie litery?
23. Jak definiuje się własny zestaw znaków?
24. Jak wydobywa się wartość pola całkowitoliczbowego?
25. Jak reprezentuje się liczby zmiennoprzecinkowe w wyrażeniach regularnych?
26. Jak wydobywa się wartość zmiennoprzecinkową z dopasowania?
27. Co to jest poddopasowanie? Jak uzyskuje się do niego dostęp?

Terminologia

dopasowanie	<code>regex_match()</code>	szukanie
<code>multimap</code>	<code>regex_search()</code>	wyrażenie regularne
podwzorec	<code>smatch</code>	wzorec

Praca domowa

1. Uruchom program analizujący zawartość wiadomości e-mail. Przetestuj go na własnym większym pliku. Pamiętaj o umieszczeniu w nim wiadomości, które mogą spowodować błędy, jak takie, które zawierają dwa wiersze adresu, kilka wiadomości o takim samym adresie lub temacie oraz puste wiadomości. Przetestuj program także na czymś, co nie jest według jego specyfikacji wiadomością, np. duży plik niezawierający wierszy ----.
2. Dodaj kontener `multimap`, w którym zapiszesz tematy. Niech program pobiera łańcuch wejściowy z klawiatury i drukuje każdą wiadomość z tym łańcuchem jako tematem.
3. Zmodyfikuj program z podrozdziału 23.4, aby znajdował temat i nadawcę za pomocą wyrażeń regularnych.

4. Znajdź prawdziwy plik poczty (zawierający prawdziwe wiadomości) i zmodyfikuj program analizujący wiadomości e-mail, aby wydobywał wiersze tematu z nazw nadawców pobieranych na wejściu od użytkownika.
5. Znajdź duży plik wiadomości e-mail (zawierający tysiące wiadomości) i zmierz czas przy zapisywaniu go w kontenerach `multimap` i `unordered_multimap`. Zwróć uwagę, że nasz program nie wykorzystuje faktu, iż kontener `multimap` jest posortowany.
6. Napisz program znajdujący daty w pliku tekstowym. Wydrukuj każdy wiersz zawierający przynajmniej jedną datę w formacie numer-wiersza: wiersz. Zacznij od wyrażenia regularnego dla prostego formatu daty, np. 12-24-2000, i przetestuj program. Następnie dodaj inne formaty.
7. Napisz program (podobny do poprzedniego) znajdujący w pliku numery kart kredytowych. Poszukaj informacji na temat formatu zapisu numerów kart kredytowych.
8. Zmodyfikuj program z punktu 23.8.7, aby na wejściu pobierał wzorec i nazwę pliku. Na wyjściu niech wysyła ponumerowane wiersze (numer-wiersza: wiersz), które zawierają dopasowanie wzorca. Jeśli nic nie zostanie dopasowane, program nie powinien nic drukować na wyjściu.
9. Za pomocą funkcji `eof()` (punkt B.7.2) można sprawdzić, który wiersz tabeli jest ostatni. Wykorzystaj ten fakt, aby (spróbować) uprościć program z podrozdziału 23.9. Pamiętaj o przetestowaniu programu na plikach kończących się pustymi wierszami za tabelą oraz takich, które w ogóle nie kończą się znakiem nowego wiersza.
10. Zmodyfikuj program z podrozdziału 23.9, aby tworzył nową tabelę, w której wiersze zawierające taką samą pierwszą cyfrę (oznaczającą rok — pierwsza klasa zaczyna się od 1) są scalone.
11. Zmodyfikuj program z podrozdziału 23.9, aby sprawdzał, czy liczba uczniów w danym przedziale lat rośnie czy maleje.
12. Na podstawie programu, który znajduje wiersze zawierające daty (zadanie 6.), napisz program znajdujący wszystkie daty i zmieniający ich format na standard ISO `rrrr-mm-dd`. Program powinien pobierać plik wejściowy i zwracać na wyjściu taki sam plik, tylko ze zmienionymi formatami dat.
13. Czy kropka pastuje do `'\n'`? Napisz program, który pozwoli to sprawdzić.
14. Napisz program, który, podobnie jak program z punktu 23.8.7, można wykorzystać do eksperymentowania z dopasowywaniem wzorców poprzez ich wpisywanie na wejściu. Ten niech jednak wczytuje plik do pamięci (reprezentując złamanie wiersza za pomocą znaku `'\n'`), aby można było przetestować wzorce zajmujące więcej niż jeden wiersz. Przetestuj go i udokumentuj około 10 wzorców testowych.
15. Opisz wzorec, którego nie da się przedstawić za pomocą wyrażenia regularnego.
16. Dla ekspertów: udowodnij, że wzorec znaleziony w poprzednim zadaniu nie jest w rzeczywistości wyrażeniem regularnym.

Podsumowanie

Łatwo można wpaść w pułapkę myślenia, iż komputery i przetwarzanie komputerowe dotyczą tylko liczb, a więc że przetwarzanie komputerowe to rodzaj matematyki. Oczywiście tak nie jest. Spójrz tylko na ekran swojego monitora, na którym widać mnóstwo tekstu i obrazów graficznych. Może akurat odtwarzana jest muzyka. Bez względu na to, co się robi, najważniejszy jest dobór odpowiednich narzędzi. W kontekście języka C++ oznacza to dobór odpowiednich bibliotek. Jeśli chodzi o przetwarzanie tekstu, często kluczowym narzędziem są wyrażenia regularne — nie zapominajmy też o słownikach i standardowych algorytmach.



