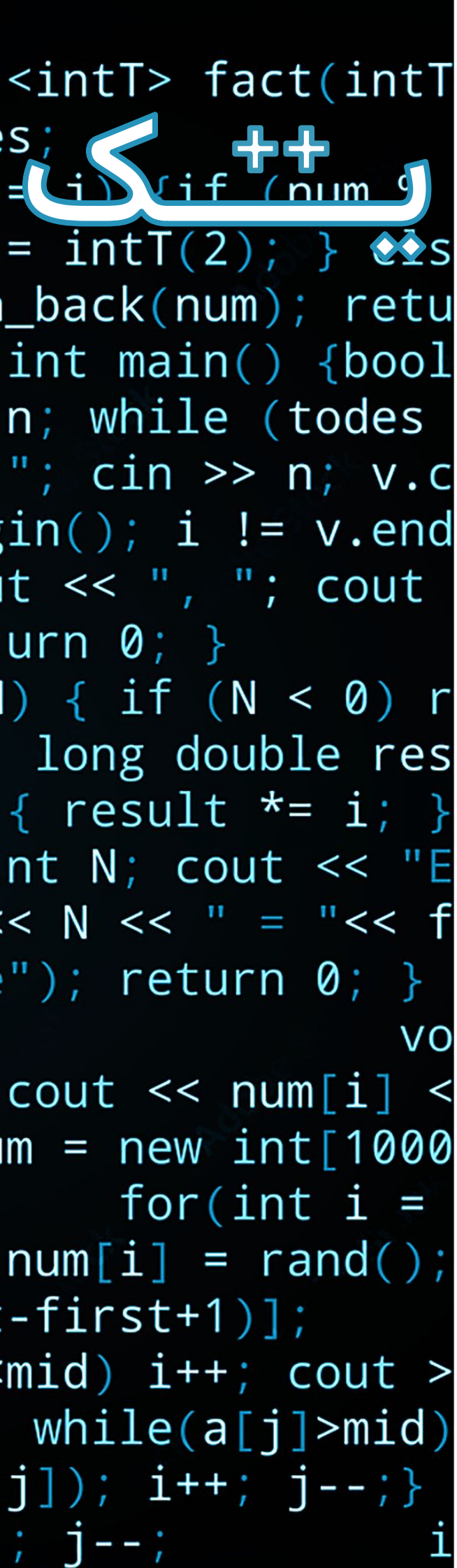

سی++ بعلاوه

```
std::cout
```

<< Regex

<< Regex in C++

```
        << std::endl;
```

دانشکده مهندسی برق و کامپیوتر دانشگاه تهران

Regex

احتمالا تا الآن با تعدادی از regular expression-ها برخورد کرده‌اید. در این مطلب قصد داریم ابتدا با تعاریف کلی و کاربردهای آن آشنا شویم و سپس به شیوه نوشتن و قواعد آن‌ها بپردازیم. در نهایت کتابخانه استاندارد regex در سی++ را معرفی می‌کنیم.

مقدمه

regular expression-ها یا به اختصار Regex، رشته‌هایی هستند که طبق قوانین و الگوهای خاصی نوشته شده و می‌توان از آن‌ها برای دسترسی راحت‌تر به اطلاعات استفاده کرد. زبان یک regex نشان‌دهنده مجموعه‌ای از رشته‌ها می‌باشد که با regex ما تطابق دارند. در ادامه با مفاهیم زبان و تطابق یک رشته با regex بیشتر آشنا میشوید. برای امتحان کردن مثال‌ها می‌توانید از این لینک یا این لینک استفاده کنید.

کاربردها

Regex-ها کاربردهای زیادی دارند. به عنوان مثال، می‌توان از آن‌ها برای موارد زیر استفاده کرد:

- پیدا کردن یک متن خاص در یک متن بزرگ‌تر.
- مقایسه رشته مورد نظر با فرمت و الگوی گفته شده (در برنامه‌های کاربردی برای صحت‌سنجی ورودی کاربر می‌توان از این قابلیت استفاده کرد).
- جایگزین کردن قسمتی از متن.
- تقسیم یک رشته به بخش‌های مختلف.

برای مثال فرض کنید به شما یک لیست از نام تعدادی فایل داده شده و شما قصد دارید نام فایل‌هایی که فرمت آن‌ها pdf است را پیدا کنید. برای این کار می‌توانید از regex زیر استفاده کنید:

```
^.+\.pdf$
```

این عبارت نام فایل‌هایی که با الگوی گفته شده تطابق دارند را پیدا می‌کند. ممکن است مثال بالا برای شما کمی نامفهوم باشد؛ در ادامه قوانین را به طور مفصل توضیح می‌دهیم تا به درک بهتری از این موضوع برسید.

Basic Syntax

در مثال‌ها، خط اول نشان‌دهنده regex، خط دوم نشان‌دهنده زیرمجموعه‌ای از L (زبان regex) و خط سوم زیرمجموعه‌ای از L' است.

به عبارت دیگر، خط دوم نشان‌دهنده رشته‌هایی می‌باشد که با regex ما مطابقت دارند و خط سوم متمم زبان L است که شامل همه رشته‌هایی می‌باشد که عضو L نمی‌باشند.

Basic Matchers

برای پیدا کردن یک رشته در یک متن بلند کافیست که همان رشته مورد نظر را تایپ کنیم. توجه کنید که regex-ها به طور پیش‌فرض case-sensitive هستند.

```
OK
{"OK"}
{"These", "strings", "won't", "match",
 "ok", ...}
```

Character Classes

این قابلیت در regex، اجازه تطابق یک کاراکتر از میان مجموعه‌ای از کاراکترها را به ما می‌دهد؛ در ادامه به توضیح این مورد می‌پردازیم.

Character Sets [ ]

با استفاده از [ ] می‌توانیم مجموعه‌ای از کاراکترهای دلخواه را انتخاب کنیم؛ کافیست کاراکترهای مد نظر را داخل براکت‌ها بنویسیم. در واقع با این کار می‌گوییم که هر یک از کاراکترهای داخل [ ] بیاید مورد قبول می‌باشد.

```
s[kpa]y
{“sky”, “spy”, “say”}
{“Say”, “sad”, “lonely”, “sKY”, ...}
```

Negated Character Sets [^]

افزودن علامت ^ به ابتدای براکت باعث می‌شود آن کاراکترها انتخاب نشوند. اگر در مثال قبل در ابتدای براکت ^ می‌گذاشتیم تمام کلمه‌هایی که شامل s در ابتدا، یک حرف در وسط و y در انتها هستند انتخاب می‌شوند به طوری که حرف وسطی k، p یا a نیست.

```
s[^kpa]y
{“sby”, “sKy”, “s_y”, ...}
{“sky”, “spy”, “say”, “lonely”, ...}
```

Letter Range ([first-last])

نوشتن علامت - (dash) درون [ ] باعث می‌شود که نیاز نداشته باشیم همه کاراکترها را جدا جدا وارد کنیم. پس با نوشتن `[first-last]` تمام کاراکترهایی که بین دو کاراکتر `first` و `second` هستند در character set ما قرار می‌گیرند. به عنوان مثال عبارت‌های زیر با یکدیگر معادل هستند:

```
[a-f] = [abcdef]
[4-9] = [456789]
[1-3c-eM] = [123cdeM]
```

توجه داشته باشید که اگر `second` پیش از `first` باشد، با خطای invalid regular expression روبرو می‌شویم.

```
b[a-f4-9]r
{“bar”, “bbr”, “bfr”, “b7r”, ...}
{“b3r”, “bgr”, “bCr”, ...}
```

Metacharacters

تعدادی character class-های از پیش تعریف شده برای مجموعه‌هایی که کاربردهای زیادی دارند وجود دارند:

```
. = [^\n]
\w = [A-Za-z0-9_]
\W = [^A-Za-z0-9_]
\d = [0-9]
\D = [^0-9]
\s = [ \t\n\r\f\v]
\S = [^ \t\n\r\f\v]
```

نقطه (.) با هر کاراکتری به جز newline تطابق می‌یابد. w مخفف word و d مخفف digit است.

از میان کاراکترهای مجموعه \s، با \n و اسپیس آشنایی دارید. بقیه نیز از انواع whitespace-ها هستند که در بخش Character Escapes توضیح داده می‌شوند. همچنین می‌توانید اینها را داخل character set-ها هم به کار ببرید:

```
[\dAbc] = [0-9Abc]
```

Options (Flags)

regex دارای option-هایی است که با استفاده از آن‌ها می‌توان نحوه تفسیر شدن را تغییر داد. مثلا می‌توان با regex نوشته شده به صورت case-insensitive برخورد کرد. فرمت مثال‌ها در این بخش به این صورت است که خط اول نشان‌دهنده ورودی و خط دوم نشان‌دهنده regex می‌باشد. کلمات پررنگ نشان‌دهنده بخش‌هایی می‌باشند که با regex تطابق دارند. توجه کنید که معمولا regex-ها را بین دو علامت / می‌گذارند.

g (global)

فلگ g باعث می‌شود همه عبارت‌های مورد قبول برگردانده شوند و نه صرفا اولین عبارت. اگر این آپشن نباشد regex در رشته ورودی جستجو کرده و صرفا اولین کلمه‌ای که تطابق داشته باشد را برمی‌گرداند.

```
sky smy sfy say sty hdsPyfd s_y s0y
/s[^kpa]y/g
sky smy sfy say sty hdsPyfd s_y s0y
/s[^kpa]y/
```

i (case-insensitive)

فلگ i باعث می‌شود بدون توجه به اینکه حروف کوچک یا بزرگ هستند عبارت مورد نظر انتخاب شود.

```
The quick brown fox jumps over the Lazy Dog
/lazy/i
The quick brown fox jumps over the Lazy Dog
/lazy/
```

m (multiline mode)

استفاده از m باعث می‌شود که regex بر روی هر کدام از خط‌ها عمل کند و نه کل متن. مثال این بخش بعد از توضیح Anchor-ها آمده است.

Quantifiers

این دسته از کاراکترها برای مشخص کردن تعداد تکرار کاراکتری که پیش از آن آمده به کار می‌روند.

Asterisk (*)

اگر بعد از یک کاراکتر علامت * را قرار دهیم، می‌گوییم که کاراکتر قبلی می‌تواند به هر تعدادی (صفر یا بیشتر) در متن بیاید.

```
bear beer br ber
/be*r/g
```

Plus Sign (+)

اگر بعد از یک کاراکتر علامت + را قرار دهیم، می‌گوییم که کاراکتر قبلی می‌تواند یک بار یا بیشتر در متن بیاید.

```
bear beer br ber
/be+r/g
```

Question Mark (?)

اگر بعد از یک کاراکتر علامت ? را قرار دهیم، می‌گوییم که آن کاراکتر قبلی می‌تواند در متن بیاید یا نیاید.

```
ruin run ruiin
/rui?n/g
```

Curly Braces ({})

برای نشان دادن اینکه می‌خواهیم یک کاراکتر دقیقا چند بار ظاهر شود، بعد از آن کاراکتر، از {n} استفاده می‌کنیم که در آن n نمایش‌دهنده تعداد دفعاتی است که می‌خواهیم آن کاراکتر ظاهر شده باشد.

```
day daay daaay daaaay
/da{3}y/g
```

اگر بخواهیم نشان دهیم که یک کاراکتر باید حداقل m بار و حداکثر n بار ظاهر شده باشد، از {m,n} استفاده می‌کنیم. اگر مقدار n را وارد نکنیم کلمه‌هایی که کاراکتر مورد نظر حداقل m بار در آن‌ها ظاهر شده انتخاب می‌شوند.

```
day daay daaay daaaay daaaaay
/da{3,}y/g
day daay daaay daaaay daaaaay
/da{3,4}y/g
```

Anchors

این دسته از کاراکترها نشان می‌دهند در کجای متن باید به دنبال رشته مورد نظر باشیم.

Dollar Sign ($)

تطابق regex و رشته، در انتهای آن بررسی می‌شود.

```
My phone number is 555-121-1231
/1\d*1$/g
```

Caret (^)

تطابق regex و رشته در ابتدای آن بررسی می‌شود.

```
Test your code, then test it again
/^test/gi

Roses are red, violets are blue
Roses are red, violets are blue
/^Roses/gm

Roses are red, violets are blue
Roses are red, violets are blue
/^Roses/g
```

Alternation

Pipe (|)

این کاراکتر همانطور که احتمالا حدس زده‌اید کار or کردن را برای ما انجام می‌دهد و تا حد زیادی به [ ] شباهت دارد. البته این دو با یکدیگر تفاوت دارند؛ عملگر [ ] برای کاراکتر استفاده می‌شود (character level) اما عملگر | را برای چند کاراکتر نیز می‌توان استفاده کرد (level expression).

```
The cat sat on the mat, is solving flat.
/(s|m|fl)at/g
```

Character Escapes

کاراکتر backslash (\) یک کاراکتر خاص است که باعث می‌شود معنای کاراکتری که بعد از آن می‌آید تغییر کند. همانگونه که دیدیم دسته‌ای از کاراکترها معنای خاصی دارند (مانند ^*|+?.[]). حال فرض کنید می‌خواهید در متن دنبال * بگردید؛ اگر به صورت عادی این کاراکتر را وارد کنید چون دارای معنی خاصی می‌باشد به صورت دیگری تفسیر می‌شود. برای این کار می‌توانید از *\ استفاده کنید که باعث می‌شود دنبال کاراکتر * در متن بگردد. البته یک سری special characters نیز وجود دارند که با \ معنی خاصی می‌گیرند.

Special Characters

کاراکترهای این دسته معمولا برای نشان دادن whitespace-ها یا موارد از این قبیل استفاده می‌شوند. تعدادی از کاراکترهای این دسته در بخش زیر لیست شده‌اند:

```
\b (word boundary)
\t (tab)
\n (newline)
\f (form feed)
\v (vertical tab)
```

Escape Characters

راهکار جستجو برای کاراکترهایی که معنای خاصی دارند استفاده از escape character یا همان \ است که باعث می‌شود به صورت کاراکتر عادی به آن نگاه شود.

```
\*
\+
\?
\.
```

برای جستجوی خود کاراکتر \ هم کافیست که آن را به صورت روبرو escape کنیم: \\

Grouping Constructs

(subexpression)

از پرانتز برای گروه کردن تعدادی regex استفاده می‌شود. گروه‌ها را می‌توان بعدا reference یا وادار به رعایت قوانینی کرد.

فرض کنید می‌خواهیم از رشته زیر اسم، سن و حرفه یک شخص را بدست آوریم:

```
Andrew Mead, 34 years old, is a Full-stack Developer
/(\w+\s\w+),\s(\d+)\syears\sold,\sis\sa\s(.*)/g
```

عبارت regex بالا با کل رشته ورودی تطابق می‌یابد ولی اسم شخص در گروه اول (\w+\s\w+)، سن شخص در گروه دوم (\d+) و حرفه فرد در گروه سوم (.*) ذخیره می‌شود. توضیحات در مورد دسترسی به محتوای گروه‌ها در بخش backreference آمده است.

(?<name>subexpression)

با نحوه گروهی کردن عبارات منظم آشنا شدیم، حال اگر بخواهیم برای گروه دلخواه اسم مشخصی مانند name را نسبت دهیم در ابتدای هر پرانتز از (?<name> یا (?'name' استفاده می‌کنیم.

فرض کنید رشته‌ای از تاریخ‌ها به صورت YYYY-MM-DD داریم و می‌خواهیم سال، ماه و روز را در گروه‌هایی از عبارات منظم به نام‌های year, month, day داشته باشیم:

```
2003-06-23
/^(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})$/
```

(?:subexperssion)

استفاده از این syntax باعث می‌شود که یک non-capturing group داشته باشیم به این معنی که نمی‌توان از خاصیت referencing برای این گروه‌ها استفاده کرد. تفاوت این گروه‌ها با گروه‌های عادی در مثال‌های backreference واضح‌تر می‌شود.

Backreference Constructs

\number

فرض کنید می‌خواهید یک عبارت regex بنویسید که در آن بخش‌هایی از رشته شما با هم برابر باشند. برای این کار می‌توانید از backreference استفاده کنید. برای backreference به گروه i-ام از \i استفاده می‌کنیم.

```
123-123-i-i
123-123-i-m
/(\d+)-\1-(\w+)-\2/g
```

یا مثلا می‌خواهیم آدرس ایمیل username و domain را بدست بیاوریم و بررسی کنیم آیا دو آدرس ایمیل ورودی دارای domain یکسانی هستند یا نه:

```
cat@meowy.com - meow@meowy.com
dog@meowy.com - woof@notmeowy.com
/([^@]+)@(.+\..+) - ([^@]+)@\2/g
```

هنگامی که می‌خواهید از backreference استفاده کنید حواستان به non-capturing group-ها باشد.

```
ab-cd,xyz-xyz
ab-cd,xyz-ab
/(?:ab)-cd,(xyz)-\1/g
```

\k<name>

اگر تعداد گروه‌ها زیاد شود، رفرنس دادن با شماره گروه کار طاقت فرسایی می‌شود پس می‌توانیم برای گروه‌ها اسم بگذاریم تا راحت‌تر به آن‌ها دسترسی داشته باشیم.

```
cat@meowy.com - meow@meowy.com
dog@meowy.com - woof@notmeowy.com
/(?<address>[^@]+)@(?<domain>.+\..+) -
([^@]+)@\k<domain>/g
```

Regex in C++

Regex library

سی++ دارای کتابخانه‌ای به نام regex است که ابزار بسیار قدرتمندی برای کار با متن‌ها است.
این کتابخانه با استفاده از عبارات منظم به ما امکان جستجو، دستکاری و صحت‌سنجی در متن‌ها را فراهم می‌کند.
چهار بخش اصلی کار با عبارات منظم به صورت زیر می‌باشند که برای هر کدام یک آبجکت داریم:

Target Sequence:

دنباله‌ای از کاراکترها که در آن جستجو می‌کنیم. این دنباله می‌تواند به صورت string سی (آرایه‌ای از کاراکترها که با نال (\0) به پایان می‌رسد) و یا به صورت std::string باشد.

Pattern:

همان regex ما است که به صورت یک آبجکت از جنس std::basic_regex می‌باشد.

Matched Array:

اطلاعات درباره تطابق‌های پیدا شده می‌توانند به صورت یک آبجکت از جنس std::match_results بازیابی شوند.

Replacement String:

متنی که تطابق‌های یافت شده را با آن تعویض می‌کنیم.

Main Classes

تمامی کلاس‌های مربوط به عبارات منظم در کتابخانه استاندارد <regex> قرار دارند.

std::basic_regex

یک کلاس template شده است که وظیفه آن نگهداری یک regex است. برای string-های عادی، از std::basic_regex<char> یا همان **std::regex** استفاده می‌کنیم.
برای ساخت یک آبجکت از آن می‌توان از کانستراکتورهای زیر استفاده کرد:

```
regex();
```

دیفالت کانستراکتور یک regex تهی ایجاد می‌کند که چیزی را match نمی‌کند.

```
regex(const std::string& pattern,
      std::syntax_option_type flags);
regex(const char* pattern,
      std::syntax_option_type flags);
```

این کانستراکتورها یک regex از رشته داده شده می‌سازند که نحوه تفسیر آن را flags مشخص می‌کند.
flag-ها می‌توانند syntax و الگوریتم regex و option-های آن را مشخص کنند.

`ECMAScript / basic / extended / awk / grep / egrep`

این flag-ها نشان می‌دهند که regex از چه syntax و الگوریتمی پیروی می‌کند. مقدار پیش‌فرض آن syntax مشهور ECMAScript است. حداکثر یکی از این نوع فلگ‌ها می‌تواند اعمال شود.

`icase`

این flag باعث می‌شود که case-insensitive رفتار کند.

`multiline`

عملکرد این flag را در قسمت‌های قبل توضیح داده‌ایم.

`nosubs`

با اعمال این flag همه non-capturing group-ها و معادل (?:subexpression) می‌شوند.

`optimize`

این flag برای بهبود عملکرد در زمان کامپایل regex استفاده می‌شود. استفاده از آن به regex engine می‌گوید که در هنگام کامپایل، بهینه‌سازی انجام دهد. این بهینه‌سازی باعث بهبود عملکرد regex هنگام جستجو می‌شود.

برای استفاده از چندین flag کافیست آن‌ها را با استفاده از عملگر (|)، or کنیم:

```
std::regex rgx("^test$",
               std::regex::icase |
               std::regex::multiline);
```

std::sub_match

یک کلاس template شده است که برای string عادی معادل **std::ssub_match** می‌شود. هر قسمت مطابقت‌داده‌شده در متن ورودی به صورت یک object از تایپ این کلاس ذخیره می‌شود.

این کلاس، جفت iterator-هایی به string اصلی دارد که بازه مطابقت regex را نشان می‌دهد. توجه کنید که ما خودمان این کلاس را کانستراکت نمی‌کنیم و صرفا توسط نتیجه الگوریتم‌های سرچ و غیره از آن استفاده می‌کنیم.

std::match_results

یک کلاس template شده است که برای string عادی معادل **std::smatch** می‌شود. این کلاس گروهی از مطابقت‌ها یعنی std::ssub_match-ها را ذخیره می‌کند که می‌توان با استفاده از اپراتور [ ] به آنها دست یافت.

اولین std::ssub_match داخل یک std::smatch (یعنی اندیس 0 آن) همواره کل match است و محتوای اولین group در اندیس 1 می‌باشد.

Algorithms

حال به توابع اصلی استفاده از regex-ها در سی++ می‌پردازیم. با استفاده از این توابع می‌توان عملیات‌های matching و replacing را انجام داد.

std::regex_match

این تابع تطابق regex با کل رشته ورودی را چک می‌کند و پاسخ را به صورت یک bool برمی‌گرداند. برخی از overload-های این تابع به صورت زیر اند:

```
std::regex_match(const std::string& str,
                 const std::regex& re);
std::regex_match(const char* str,
                 const std::regex& re);
std::regex_match(Iterator first,
                 Iterator last,
                 const std::regex& re);
```

مثال:

```
std::string input("1234");
std::regex pattern("\\d{4}");
if (std::regex_match(input, pattern)) {
   std::cout << "Match found!";
}
```

توجه کنید که برای استفاده از متاکاراکتر \d، بک‌اسلش escape شده است. این به خاطر این است که خود سی++ کاراکتر \ را خاص در نظر می‌گیرد و برای درج \ در string باید آن را escape کنیم. به طور مثال برای درج کاراکتر واقعی \ در یک regex باید \\\\ بنویسیم که در string معادل \\ می‌شود و regex آن را یک \ در می‌گیرد.

برای خاص نگرفتن \ در سی++ می‌توان از raw string literal-ها استفاده کرد که به صورت زیر اند:

```
str = R"(this is not a "newline": \n)";
```

پشت string کاراکتر R قرار می‌گیرد و کنار "-ها باید پرانتز بیاید. رشته بالا معادل زیر است:

```
str = "this is not a \"newline\": \\n";
```

overload-های دیگر این تابع، در پارامتر دوم، یک رفرنس به آبجکت std::smatch می‌گیرند که نتیجه گروه‌ها و بازه match شده در آن است:

```
regex_match(const std::string& str,
            std::smatch& m,
            const std::regex& re);
```

نحوه استفاده از این نوع overload-ها در الگوریتم بعدی توضیح داده شده است.

std::regex_search

این تابع به ازای regex مشخص‌شده، در string ورودی به دنبال مطابقت در هر قسمت آن می‌گردد و نتیجه جستجو را به صورت یک bool برمی‌گرداند. overload-های این تابع مشابه الگوریتم قبلی std::regex_match اند:

```
regex_search(const std::string& str,
             std::smatch& m,
             const std::regex& re);
```

تفاوت این تابع با std::regex_match در آن است که این تابع همه بخش‌های string ورودی را برای تطابق با regex چک می‌کند ولی در std::regex_match تمام رشته باید با regex تطابق داشته باشد.

نحوه استفاده با یک مثال:

```
int main() {
  std::string inp = "Sample text 123 here"
                    " for test.";
  std::regex pat(R"(text\s(\d+)\s(\w+))");

  std::smatch match;
  if (std::regex_search(in,match,ptrn)) {
    std::cout << "Matched input!\n";

    std::string whole_match;
    whole_match = match[0].str();
    std::cout << "Whole match: "
              << whole_match << '\n';

    int len = match.size();
    for (int i = 1; i < len; i++) {
       std::cout << "Group " << i << ": "
                 << match[i] << '\n';
    }
  }
  return 0;
}
```

خروجی به صورت زیر است:

```
Matched input!
Whole match: text 123 here
Group 1: 123
Group 2: here
```

اگر در اینجا از std::regex_match استفاده شده بود، خروجی تابع false می‌شد.

برای یافتن تمامی match-ها در یک رشته ورودی (مانند فلگ g)، باید از overload-ای که iterator می‌گیرد استفاده کرد و پس از هر match شدن در if، مقدار شروع iterator را آپدیت کرد.

std::regex_replace

این تابع اجازه می‌دهد تمام تطابق‌های موجود در ورودی را با عبارتی خاص جایگزین کنیم. خروجی این تابع string تغییر یافته است. این تابع فراخوانی‌های متفاوتی دارد که معمول ترین آن‌ها به شکل زیر است:

```
regex_replace(std::string input,
              std::regex pattern,
              std::string replacement);
```

سینتکس substitution در این مطلب گفته نشده و می‌توانید درباره آن تحقیق کنید. متاکاراکترها در آنجا با $ شروع می‌شوند و مثلا $& به معنای جایگذاری با کل match است.

مثال:

```
int main() {
  std::string txt = "Xyz Text.";
  std::regex pat("x",
                 std::regex::icase);
  std::string rpl = "REP";

  std::string res;
  res = std::regex_replace(txt,pat,rpl);

  std::cout << res;
  return 0;
}
```

خروجی به صورت زیر است:

```
REPyz TeREPt.
```

در صورتی که replacement = "M$&M" بود، خروجی MXMyz TeMxMt می‌شد.

در طول سالیان متمادی، همواره تلاش و وظیفه دستیاران آموزشی، کمک به ارائه مفیدتر درس و انتقال بهتر مطالب به دیگران بوده است؛ دستیاران آموزشی درس برنامه‌سازی پیشرفته نیز از این قاعده مستثنی نبوده و در طول ترم‌های گذشته همواره سعی کرده‌اند در قالب‌های متفاوت، به انتقال مفاهیم این درس کمک کرده باشند.

سی بعلاوه پلاس‌کی مجله‌ای در راستای همین هدف است که اولین موضوع از آن در ترم بهار 1402 ارائه شد و همچنان ادامه دارد. موضوعات این مجله فراتر از مقاصد پایه درس بوده و صرفا برای اطلاعات بیش‌تر و درک بهتر مفاهیم ارائه می‌شوند. خواندن و یادگیری آن اجباری نیست ولی برای یادگیری عمیق‌تر توصیه می‌شود.

تاریخ انتشار: بهار 1402

نویسندگان: الهه خداوردی، شهریار عطار

ویراستاران: سامان اسلامی نظری، میثاق محقق

طراحان: الهه خداوردی، شهریار عطار

دستیار آموزشی ارشد: طاها فخاریان