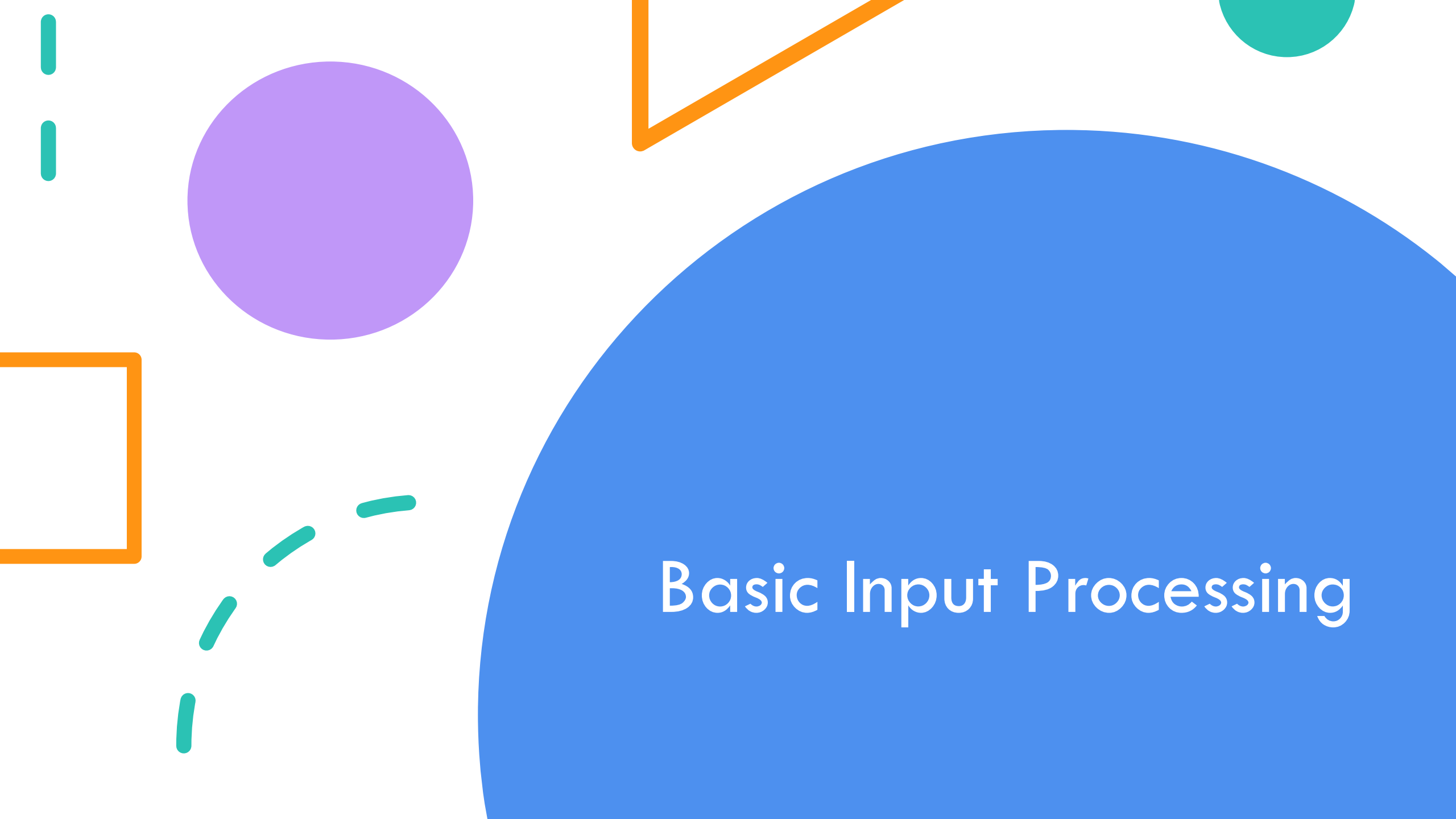

Basic Input Processing

Agenda

Input Methods in C++

Python Input Methods

Regex

Some Problems!

Final tips & takeaways

Input Methods in C/C++

	Scanf	Cin
Usage	Call a function with input string	A class with extraction Operator
Type Safety	No	Yes
Performance	Faster	Slower, due to synchronizing with C stdio's buffer
Overflow Handling	Handles overflow	Undefined behavior on overflow
Interpretation of Input	At compile time	At runtime

Scanf Guide

Code	Format
%c	Character
%d	Signed decimal integers
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)
%p	Displays a pointer
%n	The associated argument must be a pointer to an integer. This specifier causes the number of characters written so far to be put into that integer.
%i	Signed decimal integers
%e	Scientific notation (lowercase e)
%E	Scientific notation (uppercase E)
%f	Decimal floating point
%g	Uses %e or %f, whichever is shorter
%G	Uses %E or %F, whichever is shorter
%o	Unsigned octal

Compare Performance

1.00s user

3.21s system 25% cpu

16.751 total

```c
#include<cstdlib>
#include<cstdio>
int main()
{
    char buffer[256];
    while (scanf("%s", buffer) != EOF) {    }
    return 0;
}
```

3.43s user

3.70s system 29% cpu

24.521 total

```c
int main()
{
    char buffer[256];
    while (cin >> buffer)   {    }
    return 0;
}
```

Why is scanf faster than cin?

- scanf() has to explicitly declare the input type, but cin has the redirection operation overloaded using templates.
- iostream makes use of stdio's buffering system. So, cin wastes time synchronizing itself with the underlying C-library's stdio buffer, so that calls to bothscanf()and cin can be interleaved.
- How to fix it?

```
std::ios_base::sync_with_stdio(false);
```

```
#include <iostream>

using namespace std;

int main() {
   int a;
   ios_base::sync_with_stdio(false);
   cin.tie(NULL);
   cout << "Enter";
   cin >> a;
   cout << a << endl;
}
```

What does exactly happen there?

```
std::ios_base::sync_with_stdio(false);
```

- Buffer -> simply buffer is a temporary placeholder, operations are performed faster.
- Flushing -> storing buffered data to permanent memory.
- Simply it unties cin from cout which means output is flushed/displayed on the console only on demand or when the buffer is full.(AVOIDS FLUSHING)
- Also, endl, flushed the output buffer, but '\n' doesn't

A subtle problem with cin!

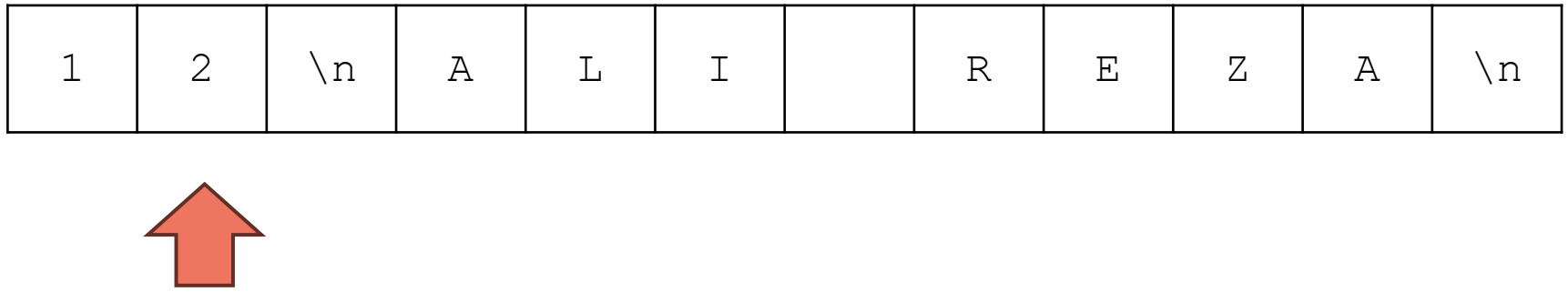
```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int age;
    string name;
    cin >> age;
    getline(cin, name);
    cout << name
         << " has age " << age << endl;
    return 0;
}
```

```
└ ./a.out
15
 has age 15
```

Why this happened?

- cin treats whitespaces as garbage! Ignore them! Leave them!
- Don’t underestimate whitespaces!
- We can assume cin terminates reading at red arrow. But we getline is called and face \n terminates reading!



How to Solve it?

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int age;
    string name;
    cin >> age;
    getline(cin.ignore(), name);
    cout << name << " has age " << age << endl;
    return 0;
}
```

Input Divide & Conquer: sstream

- Consider you want to read the input and then read each part of that later.
- Stream class to operate on strings.
- Objects of this class use a string buffer that contains a sequence of characters. This sequence of characters can be accessed directly as a string object, using member str.

Input Divide & Conquer: sstream

```cpp
#include <bits/stdc++.h>
#include <sstream>
using namespace std;

int main() {
    string s,t;
    getline(cin, s);
    stringstream w(s);

    while(getline(w, t, ' '))
        cout << t << endl;

    return 0;
}
```

```
└─ ./splitString
Programming is fun!
Programming
is
fun!
```

Python Input and String

input

`input` : Reads a string from the user's keyboard.

- reads and returns an entire line of input *

```
>>> name = input("Howdy. What's yer name?")
Howdy. What's yer name? Paris Hilton

>>> name
'Paris Hilton'
```

* *NOTE: Older v2.x versions of Python handled user input differently. These slides are about the modern v3.x of Python*

How to handle more complicated inputs

- Read two integers
- Read all floats in a line
- Read different type in a line

A Faster Way

```
import sys
def get_ints(): return map(int, sys.stdin.readline().strip().split())


a,b = get_ints()
```

```
import sys
def get_ints(): return list(map(int,
sys.stdin.readline().strip().split()))


Arr = get_ints()
```

String Processing in Python

- len(string): Returns the length of the string.
  - Example: len("Hello, World!") returns 13
- str.lower(): Converts all uppercase characters in a string into lowercase characters.
  - Example: "Hello, World!".lower() returns "hello, world!"
- str.upper(): Converts all lowercase characters in a string into uppercase characters.
  - Example: "Hello, World!".upper() returns "HELLO, WORLD!"
- str.split(sep=None): Splits a string into a list where each word is a list item.
  - Example: "Hello, World!".split() returns ['Hello,', 'World!']
- str.replace(old, new): Replaces a specified phrase with another specified phrase.
  - Example: "Hello, World!".replace("World", "Python") returns "Hello, Python!"

String Processing in Python

- str.startswith(prefix): Returns True if the string starts with the specified prefix.
  - Example: "Hello, World!".startswith("Hello") returns True
- str.endswith(suffix): Returns True if the string ends with the specified suffix.
  - Example: "Hello, World!".endswith("!") returns True
- str.find(sub): Searches the string for a specified value and returns the position of where it was found.
  - Example: "Hello, World!".find("World") returns 7
- str.join(iterable): Takes all items in an iterable and joins them into one string.
  - Example: ", ".join(["apple", "banana", "cherry"]) returns "apple, banana, cherry"
- str.strip([chars]): Returns a trimmed version of the string.
  - Example: " Hello, World! ".strip() returns "Hello, World!"

String Processing in Python

- str.count(sub): Returns the number of times a specified value occurs in the string.
  - Example: "Hello, World!".count("o") returns 2
- str.partition(sep): Searches for a specified string, and splits the string into a tuple containing three elements.
  - Example: "I love Python".partition("love") returns ('I ', 'love', ' Python')
- str.zfill(len): Fills the string with a specified number of 0 values at the beginning.
  - Example: "50".zfill(5) returns "00050"

Regular Expression

What is Regular Expression

- **regular expression** ("regex"): describes a pattern of text
  - can test whether a string matches the expr's pattern
  - can use a regex to search/replace characters in a string
  - very powerful, but tough to read

- regular expressions occur in many places:
  - text editors (TextPad) allow regexes in search/replace
  - languages: JavaScript; Java `Scanner`, `String split`
  - Unix/Linux/Mac shell commands (`grep`, `sed`, `find`, etc.)

Basic Regular Expressions

- the simplest regexes simply match a particular substring

- this is really a pattern, not a string!

- the above regular expression matches any line containing "abc"
  - *YES* : `"abc"`, `"abcdef"`, `"defabc"`, `".=.abc.=."`, ...
  - *NO* : `"fedcba"`, `"ab c"`, `"AbC"`, `"Bash"`, ...

Wildcards and anchors

`.` (a dot) matches any character except `\n`

- `".oo.y"` matches `"Doocy"`, `"goofy"`, `"LooPy"`, ...
- use `\.` to literally match a dot `.` character

`^` matches the beginning of a line; `$` the end

- `"^fi$"` matches lines that consist entirely of `fi`

`\<` demands that pattern is the beginning of a *word*;
`\>` demands that pattern is the end of a word

- `"\<for\>"` matches lines that contain the word `"for"`

- *Exercise* : Find lines in `ideas.txt` that refer to the C language.
- *Exercise* : Find act/scene numbers in `hamlet.txt` .

Special Characters

| means OR

- `"abc|def|g"` matches lines with "abc", "def", or "g"
- precedence of `^(Subject|Date)` vs. `^Subject|Date:`
- There's no AND symbol.

() are for grouping

- `"(Homer|Marge) Simpson"` matches lines containing `"Homer Simpson"` or `"Marge Simpson"`

\ starts an escape sequence

- many characters must be escaped to match them: / \ $. [ ] ( ) ^ * + ?
- `"\.\\n"` matches lines containing `".\n"`

Quantifiers (1)

`*` means 0 or more occurrences

- `"abc*"` matches "ab", "abc", "abcc", "abccc", ...
- `"a(bc)*"` matches "a", "abc", "abcbc", "abcbcbc", ...
- `"a.*a"` matches "aa", "aba", "a8qa", "a!?_a", ...

`+` means 1 or more occurrences

- `"a(bc)+"` matches "abc", "abcbc", "abcbcbc", ...
- `"Goo+gle"` matches "Google", "Gooogle", "Goooogle", ...

`?` means 0 or 1 occurrences

- `"Martina?"` matches lines with "Martin" or "Martina"
- `"Dan(iel)?"` matches lines with "Dan" or "Daniel"

Quantifiers (2)

`{min,max}` means between *min* and *max* occurrences

- `"a(bc){2,4}"` matches "abcbc", "abcbcbc", or "abcbcbcbc"

- *min* or *max* may be omitted to specify any number
  - `"{2,}"` means 2 or more
  - `"{,6}"` means up to 6
  - `"{3}"` means exactly 3

Character sets

`[ ]` group characters into a character set;
will match any single character from the set

- `"[bcd]art"` matches strings containing "bart", "cart", and "dart"
- equivalent to `"(b|c|d)art"` but shorter

- inside `[ ]`, most modifier keys act as normal characters
  - `"what[.!*?]*"` matches "what", "what.", "what!", "what?**!", ...

- *Exercise* : Match letter grades in 143.txt such as A, B+, or D- .

Character ranges

- inside a character set, specify a range of characters with -
  - `"[a-z]"` matches any lowercase letter
  - `"[a-zA-Z0-9]"` matches any lower- or uppercase letter or digit
- an initial ^ inside a character set negates it
  - `"[^abcd]"` matches any character other than a, b, c, or d
- inside a character set, - must be escaped to be matched
  - `"[+\-]?[0-9]+"` matches optional + or -, followed by ≥ one digit

Built-in character ranges

- `\b` word boundary (e.g. spaces between words)
- `\B` non-word boundary
- `\d` any digit; equivalent to `[0-9]`
- `\D` any non-digit; equivalent to `[^0-9]`
- `\s` any whitespace character; `[ \f\n\r\t\v...]`
- `\s` any non-whitespace character
- `\w` any word character; `[A-Za-z0-9_]`
- `\W` any non-word character
  - `/\w+\s+\w+/` matches two space-separated words

Regex flags

`/pattern/g`	global; match/replace all occurrences
`/pattern/i`	case-insensitive
`/pattern/m`	multi-line mode
`/pattern/y`	"sticky" search, starts from a given index

- flags can be combined:

  `/abc/gi` matches *all* occurrences of abc, AbC, aBc, ABC, ...

Regular Expression in Python (1)

- Regular expressions are in the ‘re’ package.
- Notation for patterns is slightly different from other languages – using raw string as an alternative to Regular string.

Regular String	Raw string
"ab*"	r"ab*"
"\\\\section"	r"\\section"
"\\w+\\s+\\1"	r"\w+\s+\1"

- First compile an expression (into an re object). Then match it against a string.
  - >>> import re
    >>> p = re.compile('ab*')

Regular Expression in Python (2)

- Matching a re object against a string is done in several ways.

Method/Attribute	Purpose
match()	Determine if the RE matches at the **beginning** of the string.
search()	Scan through a string, looking for any location where this RE matches.
findall()	Find all substrings where the RE matches, and returns them as a list.
finditer()	Find all substrings where the RE matches, and returns them as an *iterator*.

Regular Expression in Python (3)

- Grouping – You can retrieve the matched substrings using parentheses.
- Capturing groups are numbered by counting their **opening parentheses from left to right**. In the expression ((A)(B(C))), for example, there are four such groups:
  - ((A)(B(C)))
  - (A)
  - (B(C))
  - (C)
- Group zero always stands for the entire expression.

Regular Expression in C++

- **std::regex**:
  - The class template basic_regex provides a general framework for holding regular expressions.
  - Currently, the implementation is rather slow. It is not compiled (unlike python) and processes at runtime. So don't use it in for loops.
  - By 2016, it is 58 times slower that python regex!
  - So just use it when the load is some where else.

R-String in C++

- Raw string literals are string literals with a prefix containing R.
- They do not escape any character.
- Anything between the delimiters "(and)" becomes part of the string.

R"(\)";	"\\"
R"(\n\n\n\n)";	"\\n\\n\\n\\n"
R"(x = ""\y"")";	"x = \"\"\\y\"\""

Regex in C++

- std::regex_match
  - Determines if the regular expression e matches the entire target character sequence, which may be specified as std::string, a C-string, or an iterator pair.
  - Returns true if a match exists, false otherwise.

```
void basicRegexMatch(string str) {
    regex b(R"(\d{1,5})");
    cout << "Match: " << regex_match(str, b) << endl;
}
```

Regex Result in C++

- The class template std::match_results holds a collection of character sequences that represent the result of a regular expression match.

```
std::regex re("Get|GetValue");
std::cmatch m;
std::regex_match ("GetValue", m, re);  // returns true, and m[0] contains "GetValue"
std::regex_match ("GetValues", m, re); // returns false
```

Regex Result in C++

- The class template std::match_results holds a collection of character sequences that represent the result of a regular expression match.

```
std::regex re("Get|GetValue");
std::cmatch m;
std::regex_match ("GetValue", m, re);  // returns true, and m[0] contains "GetValue"
std::regex_match ("GetValues", m, re); // returns false
std::regex_search("GetValue", m, re);  // returns true, and m[0] contains "Get"
std::regex_search("GetValues", m, re); // returns true, and m[0] contains "Get"
```

SMS Polling

Read Input Part-1

```
int main() {
    int t;
    while(cin >> t) {
        if (t == 0)
            break;
        cin.ignore()
        for(int i = 0; i < t; i++) {
            string query;
            getline(cin, query);
        }
    }
}
```

Part-2: Split two parts

```
void separate(const string &query, string &phoneNumber, string &vote) {
    stringstream qs(query);
    getline(qs, phoneNumber, ':');
    getline(qs, vote);
}
```

```
string phoneNumber, vote;
separate(query, phoneNumber, vote);
```

Remove Dashes

```
void removeChar(string &str, const char c) {
    auto newEnd = remove(str.begin(), str.end(), c);
    str.erase( newEnd, str.end());
}
```

Define Regexes

```
regex nr(R"(\+?(\d{1,3})(\(?\d{1,3}\)?|0)(\d{3,8}))");
regex lr(R"(((0\d{1,3})|\(\d{1,3}\)\d{3,8}))");
regex tr(R"((\d{3,8}))");
```

Make Essential Changes

```
if(regex_match(str, nr)) {
    removeExtras(str);
}
else if(regex_match(str, lr)) {
    str.erase(0, 1);
    removeExtras(str);
    str = "98" + str;
}
else if(regex_match(str, tr)) {
    str = "9821" + str;
}
```

Check Votes

```
int checkVote(string& str) {
    if(str == "1" or str == "2" or str == "3" or str == "4")
        return atoi(str.c_str()); // stoi( str )
    else -1;
}
```

What if numbers were not unique?

```
smatch m;
regex_match(str, m, nr);
for (size_t i = 1; i < m.size(); ++i) {
    cout << "FOUND\n";
    std::cout << "Group " << i << ": " << m[i] << '\n';
}
```

- https://www.geeksforgeeks.org/fast-io-for-competitive-programming/
- https://dzone.com/articles/introduction-to-regular-expression-with-modern-c