

תכנות מתקדם ושפת C++ מצגת 6

מחרוזות וביטויים רגולריים

מחרוזת

- בשפת C מחרוזת היא מערך של תווים שמסתיים בתוו שערך 0
- בשפת C אין מחלקות אין פונקציות חברות ואין העמסת אופרטורים, וכדי לעבד מחרוזות משתמשים בפונקציות שמקבלות מחרוזת כפרמטר:

`strcpy()`, `strcat()`, `strcmp()`, `strlen()`

- בשפת C++ נשתמש במחלקה `string` שמאפשרת לעבד מחרוזות בצורה נוחה:

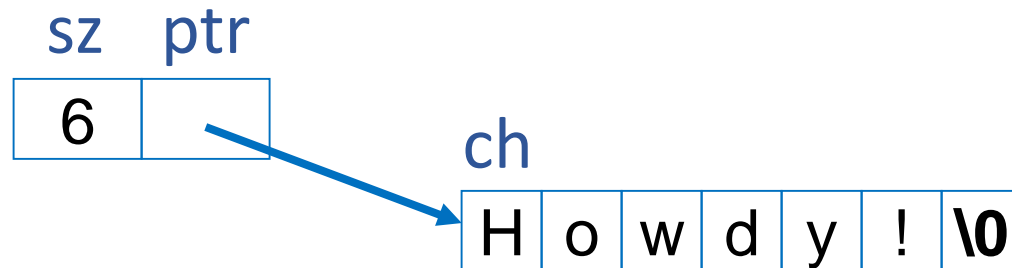
```
s1 = s2           // Assign s2 to s1
s1 += x           // Add a character or a string at end
s1 + s2           // Concatenation
s1 == s2          // Comparison
s1 < s2           // Lexicographical comparison
s.size()          // Number of characters
s.c_str()         // C-style version of string
```

מימוש המחלקה מחרוזת

- נשתמש במערך של תווים שמסתיים ב-0 לייצוג המחרוזת בתוך המחלקה, זה מפשט העתקה ממחרוזת שמיוצגת בסגנון C
- כדי ליעל, מחרוזות קצרות יישמרו בתוך האובייקט, מחרוזות ארוכות יישמרו בזיכרון הדינמי
- מחרוזת קצרה בתוך האובייקט תשמר במערך `ch`
- המשתנה `short_max` מכיל את הגודל המירבי (15) לשמירת המחרוזת בתוך האובייקט
- בשני המקרים, המשתנה `ptr` יצביע על התוו הראשון במחרוזת
- כפי שעשינו במימוש וקטור, גם כאן נקצה זיכרון עודף כדי ליעל הוספה של תווים

ייצוג המחרוזת

```
class String {  
private:  
    static const int short_max = 15;  
    int sz; // number of characters  
    char* ptr;  
    int space; // unused space on free store  
    char ch[short_max+1]; // leave space for 0 (16)  
};
```



מימוש בסיסי של מחרוזת

```
class String {  
public:  
    String() ; //default constructor: x{""}  
    String(const char* p) ; // C-style: x{"Euler"}  
    String(const String&) ; // copy constructor  
    String& operator=(const String&) ; // copy assignment  
    String(String&& x) ; // move constructor  
    String& operator=(String&& x) ; // move assignment  
    ~String() { if (sz > short_max) delete[] ptr; }  
    const char* c_str() { return ptr; } // C-style access  
    int size() const { return sz; } // number of elements  
};
```

מימוש הבנאים

- The default constructor defines a String to be **empty**:

```
String::String() : sz{0}, ptr{ch}
{
    ch[0] = 0; // terminating 0
}
```

- The constructor that takes a **C-style** string argument:

```
String::String(const char* p) : sz{strlen(p)},
    ptr{ (sz<=short_max) ? ch : new char[sz+1]},
    space{0}
{
    strcpy(ptr,p); // copy characters into ptr from p
}
```

ביטויים רגולריים

- ביטוי רגולרי הוא מחרוזת המתארת תבנית של טכסט
- הפונקציות הבאות ב- C++ מאפשרות שימוש בביטויים רגולריים:
- `regex_match` פונקציה המנסה להתאים את הביטוי הרגולרי לכל המחרוזת
- `regex_search` פונקציה המנסה להתאים את הביטוי הרגולרי לחלק מהמחרוזת
- `regex_replace` פונקציה המחליפה מופע של הביטוי הרגולרי בטכסט אחר

regex_match()

```
string input;
//regex pat("abc");
//regex pat("\\d+"); // String literals: "a\\n"
regex pat(R"(\d+)"); // Raw string literals: R"(a\n) "
//regex pat(R"(1+1=2)"); // would match 1111=2
while (true) {
    cout<<"Enter text:"<<endl;
    if(!(cin >> input)) break;
    if(regex_match(input, pat)) cout<<"Match"<<endl;
    else cout<<"No Match"<<endl;
}
```


Regex meta-characters

- There are 12 punctuation characters that make regular expressions work their magic, they are called **meta-characters**
- Any regular expression that **does not** include any of the 12 meta-characters `$()*+.?[\^{|` simply **matches itself**
- If you want your regex to match them literally, you need to escape them by placing a backslash in front of them
Thus, the regex: `*\+\.\?` matches the text `*+.`
- Those backslashes may need to be **doubled** up to quote the regex as a literal string in source code (unless you use **raw** string):

`"*"`

Regex meta-characters

- Absent from the list are the closing square bracket], the hyphen -, and the closing curly bracket }
- The first two become metacharacters only after an unescaped [, and the } only after an unescaped {
- The rules about which characters are different inside a character class:
 - **dot** is a meta character outside of a class, but not within one
 - **dash** is a meta character within a class (between two characters), but not outside
 - **caret** has one meaning outside,
 - and another meaning if specified inside a class immediately after the opening [

Match any one of several characters

- The notation using square brackets is called a **character class**
- A character class matches a **single character** out of a list of possible characters
 - Match english vowels: **[aeiouy]** (y in sky)
 - Match all common misspellings of calendar: **c[ae]l[ae]nd[ae]r**
- A hyphen (**-**) creates a range when it is placed between two characters
 - Match hexadecimal character: **[a-fA-F0-9]**
- A caret (**^**) negates the character class if you place it immediately after the opening bracket
 - **[^aeiouy]** not an English vowel
 - **[a^aeiouy]** an English vowel or **^**
 - Match non-hexadecimal character: **[^a-fA-F0-9]**

One of several characters shorthand

- Six regex tokens that consist of a backslash and a letter form **shorthand character classes**: `\d`, `\D`, `\w`, `\W`, `\s` and `\S`:
 - `\d` matches a single digit
 - `\D` matches any character that is not a digit, and is equivalent to `[^\d]`
 - `\w` matches a single word character, usually it is identical to `[a-zA-Z0-9_]`
 - `\s` matches any whitespace character - spaces, tabs, and line breaks
 - `\S` matches any character not matched by `\s`

Examples:

`\d-\d` matches **1-2, 3-4**

`\w\w-\d\d` matches **Ab-12, 12-34** - digits are in `\w`

Match any character

- **dot .** matches any character except line breaks
- **`[\s\S]`** matches any character including line breaks
- **`[\d\D]`** and **`[\w\W]`** have the same effect
- Dot abuse:

`\d\d.\d\d.\d\d` is **not a good** way to match a date

It does match **`05/16/08`**,

but it also matches **`12345678`**

Replacing the dot with a more appropriate character class

`\d\d[-./]\d\d[-./]\d\d`

allows a **hyphen, forward slash or dot**, to be used as the date separator.

Repeat part of the regex

- ***** (**star**) after a regex token means **zero or more**, example `\d*`
- **+** (**plus**) after a regex token means **one or more**, example `\d+`
- The quantifier `{n}`, repeats the preceding regex token **n** number of times
- The quantifier `{n,m}`, repeats the preceding regex token **n** to **m** times
- A **question mark ?** after a regex token means **zero or once**
- **Examples:**
 - `[A-Za-z_][A-Za-z_0-9]*` an identifier in a programming language
 - `0[xX][A-Fa-f0-9]+` C-style hexadecimal number
 - `10{100}` a googol (10^{100})
 - `[A-Fa-f0-9]{1,8}h?` 1-8 digit hexadecimal number with an **optional h** suffix
 - `colou?r` matches both colour and color
 - `A*B+C?` matches `AAABBB`, `BC`, `B` **does not match** `AAA`, `AABBCC`
 - `\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}` matches an IP Address

greedy and lazy matches

- To match an HTML tag:

This is a first test

you may attempt to use

<.+>

but this matches **first**, not ****

- The reason is that *****, **+** and **{n,m}** are **greedy**, they repeat the preceding token as many times as possible (longest match)
- You can make them **lazy** instead of **greedy** by putting a **?** after them
<.+?>
or use the pattern: **<[>]+>**
- **(ab)+** matches all of **ababab**, however **(ab)+?** matches only the first **ab**
- **.*[0-9]+** applied to **"Copyright 2003"**, **[0-9]+** matches only **"3"**

Match at start and end of a line

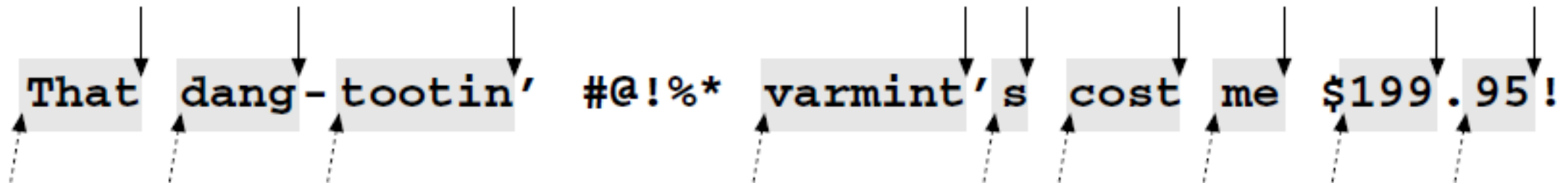
- The regular expression tokens **^**, **\$** are called **anchors**
- They do not match any characters, instead they match at certain positions, effectively anchoring the regular expression match
- **^ (caret)** matches only if it occurs at the beginning of a string
- **\$ (dollar)** matches only if it occurs at the end of a string
- **Examples:**
 - ^cat\$** a line that consists of only cat
 - ^\$** an empty line

Match whole words

- Create a regex that matches **cat** in **A cat** and **a mouse**, but not in **category** or **bobcat**

Place the word **cat** between two word boundaries **\bcat\b**

- The regular expression token **\b** is called a word boundary, it matches at the start or the end of a word
- The first **\b** requires the **c** to occur at the very start of the string, or after a non-word character.
- The second **\b** requires the **t** to occur at the very end of the string, or before a non-word character



Match one of several alternatives

- The **vertical bar or pipe symbol |**, splits the regular expression into multiple alternatives
- **Mary|Jane|Sue** matches Mary, or Jane, or Sue with each match attempt
- The regular expression finds the **leftmost match**:
When you apply **Mary|Jane|Sue** to
Jane, Mary and Sue went to Mary's house
the match Jane is found first
The match that begins earliest (leftmost) wins
- Each alternative is checked in a left-to-right order:
Jane|Janet matches **Jane** in **Her name is Janet**

Group parts of the match

- Improve the regular expression for matching Mary, Jane, or Sue by forcing the match to be a whole word
- Use grouping to achieve this with one pair of word boundaries for the whole regex, instead of one pair for each alternative
- `\b(Mary|Jane|Sue)\b` has three alternatives: Mary, Jane, and Sue, all three between two word boundaries
 - This regex **does not match** anything in **Her name is Janet**
- The alternation operator, has the **lowest** precedence of all regex operators
 - If you try `\bMary|Jane|Sue\b`, the three alternatives are `\bMary`, `Jane`, and `Sue\b`
 - This regex matches **Jane** in **Her name is Janet**

Group Parts of the Match

- Examples
- **Nov(ember)?** matches November and Nov (**greedy**)
- **Feb(ruary)? 23(rd)?** matches many alternatives
- **\b(one|two|three)\b** Find a line containing certain words:
- **(\s|:|,)*(\d+)** spaces, colons, commas followed by a number
- **<HR(+SIZE *= *[0-9]+)? *>** <HR SIZE=30>
- **\\$[0-9]+(\.[0-9][0-9])?** Dollar amount with optional cents
- We have seen two uses for grouping parentheses:
 - to limit the scope of alternation
 - to group multiple characters into larger units to which you can apply quantifiers like question mark and star

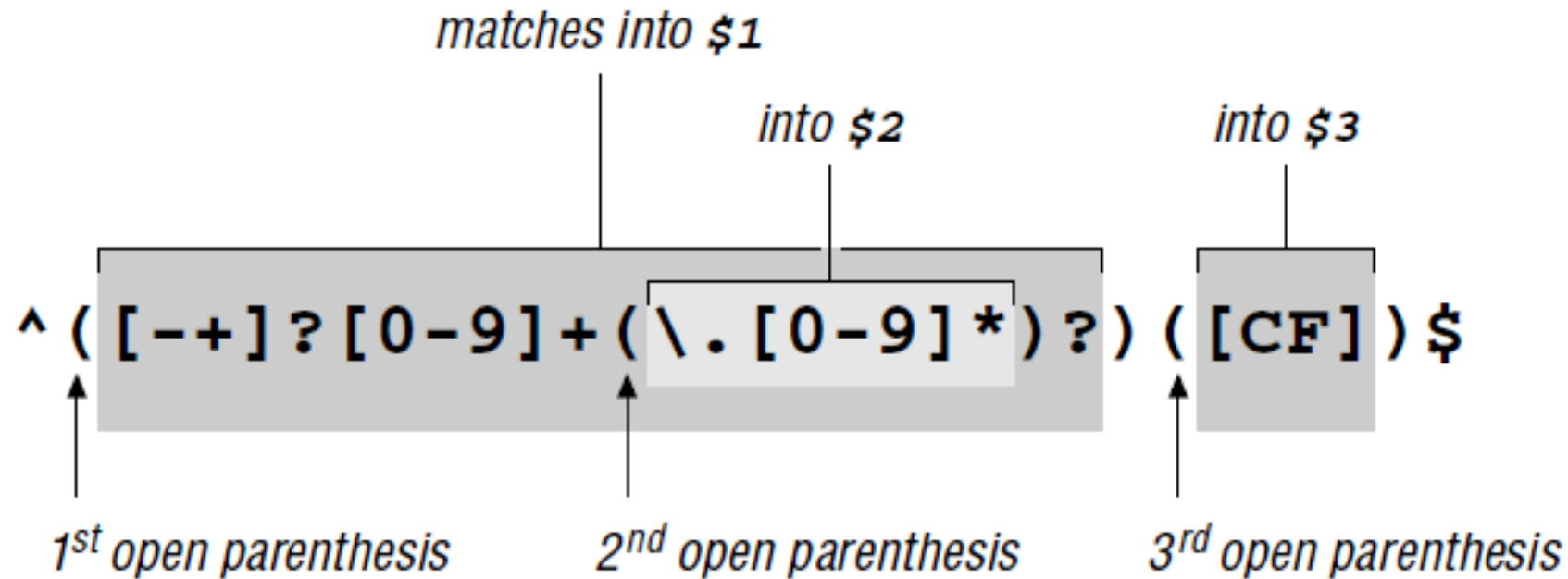
Capture parts of the match

- Create a regular expression that matches any date in **yyyy-mm-dd** format, and separately **captures** the **year**, **month**, and **day**
- A pair of **parentheses** isn't just a group, it's a **capturing** group
- Captures become useful when they cover only part of the regular expression, as in `\b(\d\d\d\d)-(\d\d)-(\d\d)\b`
- The regex `\b\d\d\d\d-\d\d-\d\d\b` does exactly the same, but does not capture
- Captures are numbered by counting opening parentheses from left to right
- There are three ways you can use the captured text:
 - match the captured text again within the same regex match
 - insert the captured text into the replacement text
 - The program can use the parts of the regex match

Capture parts of the match

- Example, convert temperatures:

`^([-+]?[0-9]+(\.[0-9]+)?)([CF])$`



- or group but do not capture:

`^([-+]?[0-9]+(?\.[0-9]+)?)([CF])$`

- Now, the text that the parentheses surrounding [CF] match, goes to \$2

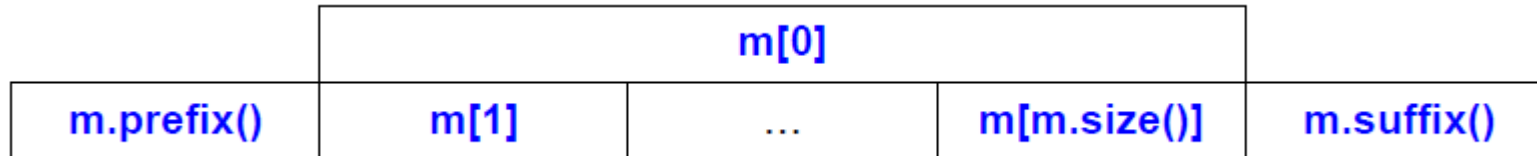
Match previously matched text again

- Create a regular expression that matches “magical” dates
- A date is magical if the year minus the century, the month, and the day of the month are all the same numbers
- For example, 2010-10-10 is a magical date:
we first have to capture the previous text, then we match the same text using a **back-reference**
`\b\d\d(\d\d)-\1-\1\b`
The `(\d\d)` matches 10, and is stored in capturing group 1
The **back-reference** `\1` matches the **10** of the **month** and **day**
- Match a pair of opening and closing HTML tags:
`<([A-Z][A-Z0-9]*)[^>]*>.*?</\1>`
- Checking for Doubled Words (the the):
`\b(\w+)\s+\1\b`

```

string input;
regex_search()
regex pattern(R"(\d+)");
smatch result;
while (true) {
    cout<<"Enter:"<<endl;
    if(!(cin >> input)) break;
    if(regex_search(input, result, pattern)) {
        cout<<"Match prefix: "<<result.prefix()<<endl;
        cout<<"Match string: "<<result[0]<<endl;
        cout<<"Match suffix: "<<result.suffix()<<endl;
    }
    else cout<<"No Match"<<endl;
}

```



regex_search()

```
void use() {
    ifstream in("file.txt"); if (!in) cerr << "no file\n";
    regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"}; //postal code
    int lineno = 0;
    smatch matches; // matched strings go here
    for (string line; getline(in,line);) {
        ++lineno;
        if (regex_search(line , matches, pat)) {
            cout << lineno << ": " << matches[0] << '\n';
            if (1 < matches.size() && matches[1].matched)
                cout << "\t: " << matches[1] << '\n'; // sub-match
        }
    }
} // TX77845 and DC 20500-0001 match
```

regex_replace()

```
string input {"x 1 y2 22 zaq 34567"};  
regex pat {"(\\w+)\\s(\\d+)"}; // word space number  
string format {"{$1,$2}\\n"};  
cout << regex_replace(input,pat,format);
```

- The output is:

```
{x,1}
```

```
{y2,22}
```

```
{zaq,34567}
```

sregex_iterator

```
regex reg("[A-Za-z]+) \\1");
string target = "the the cow jumped over over the fence";
sregex_iterator reg_begin =
    sregex_iterator(target.begin(), target.end(), reg);
sregex_iterator reg_end = sregex_iterator();
for (sregex_iterator it = reg_begin; it != reg_end; ++it) {
    cout << "Substring: " << it->str() << ", ";
    cout << "Position: " << it->position() << endl;
}
cout << "Found: " << distance(reg_begin, reg_end) << endl;
```

The output is:

```
Substring: the the, Position: 0
Substring: over over, Position: 19
Found: 2
```

Special Characters

Regular Expression Special Characters

.	Any single character (a “wildcard”)	\	Next character has a special meaning
[Begin character class	*	Zero or more (suffix operation)
]	End character class	+	One or more (suffix operation)
{	Begin count	?	Optional (zero or one) (suffix operation)
}	End count		Alternative (or)
(Begin grouping	^	Start of line; negation
)	End grouping	\$	End of line

Repetition

{ n }	Exactly n times
{ n, }	n or more times
{n,m}	At least n and at most m times
*	Zero or more, that is, {0,}
+	One or more, that is, {1,}
?	Optional (zero or one), that is {0,1}

Character Class

\d	A decimal digit
\s	A space (space, tab, etc.)
\w	A letter (a-z) or digit (0-9) or underscore (_)
\D	Not \d
\S	Not \s
\W	Not \w