Strings Matter

- All data can be represented by text
  - Books, articles, web pages
  - Tables of structured information (e.g., XML, JSON)
  - Email, SMS, social media
  - Graphics (e.g., vector formats)
  - Software code(!)
  - Binary data (e.g., uuencode, uudecode)
  - All languages and way too many emojis
- Text is very portable (except that annoying \n, \r, \r\n thingie)
- Text is easily created and edited by your choice of text editor

Options for Representing Strings

- Old C-style strings (zero-terminated char array)
  - #include <cstring> (or <string.h>)
  - strlen(s)
  - strcmp(s1, s2)
- Std::string (a class!)
  - #include <string>
  - s.size()
  - s1 == s2
- Std::basic_string (a template!!!)
  - A template for making string-like classes
  - Std::string is simply `typedef std::basic_string<char> string;`
  - Std::basic_string<wchar_t> is the same, for 16-bit chars, etc.
- Proprietary types, e.g., gtkmm's Glib::ustring (good for Unicode!)

Translating Objects to Strings

- string pi = to_string(3.14159265)
- to_string can be instanced from a simple template
for any class

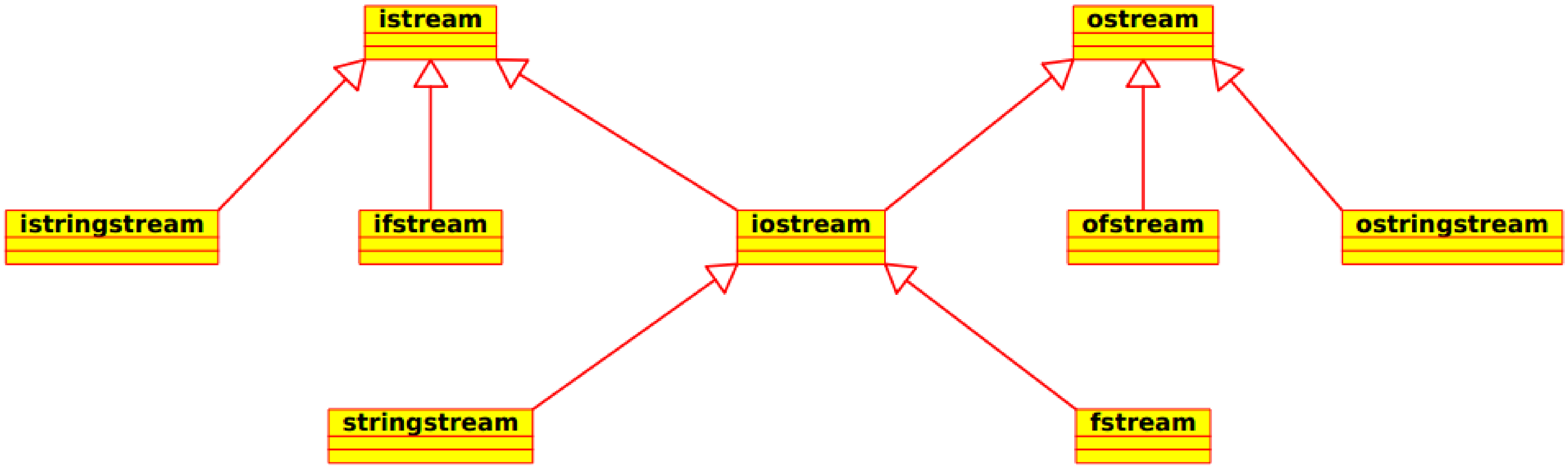
```
template<class T>
string to_string(const T& t) {
    ostringstream os;
    os << t;
    return os.str();
}
class Coordinate {
  public:
    Coordinate(int x, int y) : _x{x}, _y{y} { }
    int x() const {return _x;}
    int y() const {return _y;}
  private:
    int _x, _y;
};
ostream& operator<<(ostream& os, const Coordinate& c) {
    os << "(" << c.x() << "," << c.y() << ")";
    return os;
}
int main() {
    Coordinate c{3,4};
    string s = to_string<Coordinate>(c);
    cout << s << endl;
}
```

Translating Strings to Objects

- stoX (where X is a primitive)
  - int i = stoi(“42”);
  - double e = stod(“2.71828”);
- A template again can be instanced to take care of classes

```
template<class T>
T stox(const string& s) {
    istringstream is(s);
    T t;
    if (!(is >> t)) throw bad_from_string{};
    return t;
}
// …
    Coordinate c = stox<Coordinate>(“(3,4)”);
```

String I/O (Partial) Class Hierarchy



Regular Expressions

- A **regular expression** (regex) is a string that defines a search or search-and-replace pattern for other strings
- Two standards exist, and C++ supports both simultaneously
  - The Perl standard dates back to the 1950s, first proposed by American mathematician Stephen Cole Kleene, implemented in 1968 in the original Unix text editor QED, and enhanced in 1986 for Perl
  - The POSIX standard with more verbose but flexible syntax was adopted in 1992
- A regex is often an excellent choice for text manipulation, command or data parsing, or input validation
  - Regex are built into Perl and Javascript, and available as standard classes in C++, C#, Java, and Python. They are not part of ANSI C.
- The regex syntax is terse, cryptic, and borderline write-only, yet exceeding useful – learn it!

Special vs Non-Special Characters

- A non-special character matches itself
- A special character will match 0 or more characters
  - For example, . will match any one character, while .+ will match one or more characters
  - \s is a special character that matches a space or tab, while \s* matches zero or more spaces and tabs
  - A special character preceded by a \ matches itself, so \. matches a period and \\ matches a single backslash
  - In C++ strings (other than raw strings), another \ is required to escape the \ in the regular expression
    - So \s* would be "\\s*" in a C++ string

Special Characters

- . matches any single character except a newline or carriage return
- [ ] matches any single character inside the brackets
  - [^] matches any character NOT inside the brackets
  - [a-z] matches any characters in the range a to z
  - [cat|dog] matches cat OR dog
- {m,n} matches the previous element at least m but no more than n times
  - * matches the previous element 0 or more times (same as {0,})
  - ? matches the previous element 0 or 1 times (same as {0,1})
  - + matches the previous element 1 or more times (same as {1,})
- ^ matches the start and $ the end of a line
- ( ) forms a sub-match for later reference
  - \N matches the Nth sub-match's actual text (start at 1, NOT 0!)

Character Matching Examples

- a.c matches abc, acc, a/c
- r[au]n matches ran or run
- [^2-9][0-9] matches 17 or 03 or K9, but not 21 or 99
- 3\.3* matches 3., 3.3, 3.33, or 3.3333333333
- ([0-9]+) = \1 matches 3 = 3 and 42 = 42, but not 21 = 39
- ^[hH]ello matches hello or Hello, but only at the start of a line
- Sincerely(yours)?,$ matches Sincerely, or Sincerely yours, but only at the end of a line
- Subject: (F[Ww]:|R[Ee]:)?(.*) matches an email subject line for forwards and replies only
- [a-zA-Z] [a-zA-Z_0-9]* matches a C++ identifier

Character Classes
(C++ supports both syntax families)

- Unix / Linux traditional syntax
  - \s matches a whitespace char
    - \S anything else
  - \w matches a word char ([A-Za-z0-9_])
    - \W anything else
  - \d matches a digit ([0-9])
    - \D anything else
  - \x represents a hexadecimal digit
    - \X anything else
  - \n, \r, \t represent newline, carriage return, and tab as usual

- ECMA class syntax

[:alnum:]	alpha-numerical character
[:alpha:]	alphabetic character
[:blank:]	blank character
[:cntrl:]	control character
[:digit:]	decimal digit character
[:graph:]	character with graphical rep
[:lower:]	lowercase letter
[:print:]	printable character
[:punct:]	punctuation mark character
[:space:]	whitespace character
[:upper:]	uppercase letter
[:xdigit:]	hexadecimal digit character
[:d:]	decimal digit character
[:w:]	word character
[:s:]	whitespace character

http://www.cplusplus.com/reference/regex/ECMAScript/

More Character Matching Examples

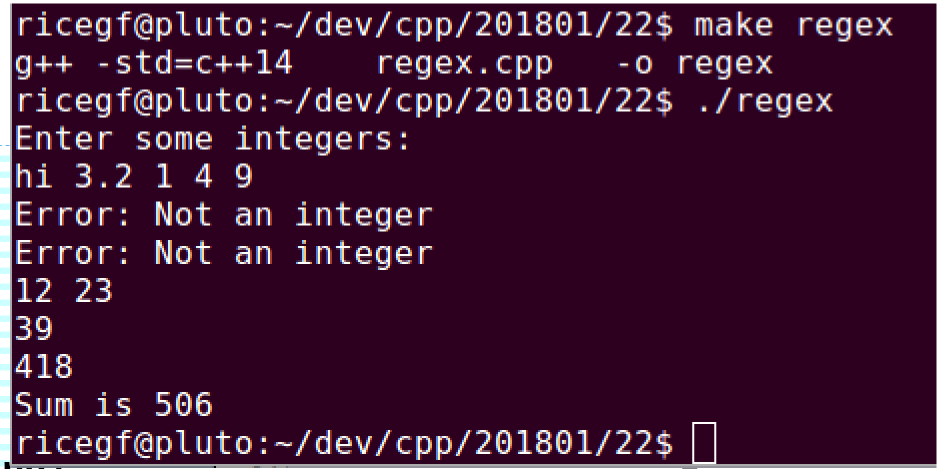
- Xa{2,3}   matches Xaa   Xaaa
- Xb{2}   matches Xbb
- Xc{2,}   matches Xcc   Xccc   Xcccc   Xccccc
- int\s+x\s*=\s*\d+; matches an integer definition
- CSE\d{4} matches a CSE class
- \w{2}-\d{4,5}   matches 2 letters and 4 or 5 digits
- \d{5}(-\d{4})? matches a 5 or 9 digit zip code
- [^aeiouy]   matches not an English vowel

Some C++ Regex-Related Classes and Functions

- The regex class is constructed with the regular expression string as a parameter
  - If the regular expression has syntax errors, the constructor throws a regex_error exception
  - Once instanced, the regex may be used in multiple matches and searches
- regex_match(string_to_search, regex) returns true if the entire string_to_search matches the regex
- regex_search(string_to_search, regex) returns true if any substring of string_to_search matches regex

http://www.cplusplus.com/reference/regex
/

A Simple C++ Regex Example



```
ricegf@pluto:~/dev/cpp/201801/22$ make regex
g++ -std=c++14     regex.cpp    -o regex
ricegf@pluto:~/dev/cpp/201801/22$ ./regex
Enter some integers:
hi 3.2 1 4 9
Error: Not an integer
Error: Not an integer
12 23
39
418
Sum is 506
ricegf@pluto:~/dev/cpp/201801/22$ 
```

```
#include <iostream>
#include <regex>
#include <string>

int main() {
    std::string input;
    std::regex integer{"(\\+|-)?[[:digit:]]+"};
    std::cout << "Enter some integers:" << std::endl;

    int sum = 0;
    while(std::cin>>input) {
        if(std::regex_match(input,integer))
            sum += stoi(input);
        else
            std::cerr << "Error: Not an integer" << std::endl;
    }
    std::cout << "Sum is " << sum << std::endl;
}
```

Regex are often VERY useful for data validation!

Regex and Beyond

- Regex are extremely powerful text manipulators
  - We’ve only scratched the thin outer surface
- Regex are widely used outside C++
  - The gedit text editor can search (and replace) using regex
  - Command line tools like grep (literally “global regular expression print”), find, and awk use regex
  - Perl famously extends regex power to ludicrous (and incredibly useful!) extremes
- If you deal with much text, you need to learn regex!