4 Strings and Streams. Testing.

Objective: to practice using the standard library string and stream classes.

Read:

- Book: strings, streams, function templates, exceptions.

1 Unit testing

When writing code, testing is important and a common methodology in modern *agile* software development methodologies is *test driven design* or "test first". One major benefit of writing tests before writing the code is that this helps with understanding the problem and structuring the code, starting from the desired result and working ones way towards the solution. That is an application of the principle of *programming by intention* (or "wishful thinking") and helps designing functions with suitable parameters and return types.

In the strict formulation, you are only allowed to write new code if there is a unit test that doesn't pass. So to add new functionality you first write a unit test and make the test call the desired new function. In doing so, you specify both what the arguments to the function will be, and what type and value the function should return for the given arguments. This also includes defining any new types that you need to express the desired functionality. At this stage, your test will not compile, so now you add an empty function (with `return 0;`, `return false;` or what is suitable). Run the tests and make sure that they fail — if not, either the functionality is already supported or the test case is wrong. Then, implement the function to make the test case pass.

Another good testing principle is to write *unit tests*, that tests "the smallest testable unit" (e.g., functions, classes) in addition to large-scale tests that verifies the funciton of the entire system. Unit tests are valuable during development as they make it easy to check if additions to the system have affected the behaviour of the old (apparently unrelated) functionality. If the unit tests ran successfully before a modification to the code, they should also run successfully after it.

It is also often a good idea to write test programs that do not require manual user input or manually checking the output. For two small examples of such test programs, study the examples given in lab1: `test_coding` and `test_editor`.

A0. Read through the assignments of this lab (A1 – A5), and write test programs for each assignment. For instance, for the first assignment (A1), create a file (or string) where you have manually removed the HTML tags from the given HTML file and write a program that calls your tag removal function and compares its result with the manually created file (string). Start with smaller unit tests, e.g. for testing the removal of HTML tags and the replacement of special characters. For those, write small test cases that test just one thing ("unit of functionality").

For the second assignment (A2), you can translate the example with the numbers 0–35 into code, tetsting that your functions returns both the correct strings (`"CCPPC..."` and the corresponding prime number sequence).

For the third assignment (A3), there is a main program (that requires user input) given. You can base your test program on that. A good option here is to give the streams to use as parameters instead of hard-coding `std::cin` and `std::cout`. Then, one can use `std::stringstream` to automate both the input and checking the results. (See section 3.2 of this lab for info on stringstreams.)

For the last assignment (A5), remember to also test that your function throws exceptions correctly.

2 Class string

2.1 Introduction

In C, a string is a null-terminated array of characters. This representation is the cause of many errors: overwriting array bounds, trying to access arrays through uninitialized or incorrect pointers, and leaving dangling pointers after an array has been deallocated. The `<cstring>` library contains operations on C-style strings, such as copying and comparing strings.

C++ strings hide the physical representation of the sequence of characters. The exact implementation of the `string` class is not defined by the C++ standard.

The `string` identifier is not actually a class, but a type alias for a specialized template:

```
using string = std::basic_string<char>;
```

This means that `string` is a string containing characters of type `char`. There are other string specializations for strings containing "wide characters". We will ignore all "internationalization" issues and assume that all characters fit in one byte.

`string::size_type` is a type used for indexing in a string. `string::npos` ("no position") is a value indicating a position beyond the end of the string; it is returned by functions that search for characters when the characters aren't found.

2.2 Operations on Strings

The following class specification shows most of the operations on strings:

```
class string {
public:
    /*** construction ***/
    string();                 // creates an empty string
    string(const string& s); // creates a copy, also has move constructor
    string(const char* cs);  // creates a string with the characters from cs
    string(size_type n, char ch); // creates a string with n copies of ch

    /*** information ***/
    size_type size();         // number of characters

    /*** character access ***/
    const char& operator[](size_type pos) const;
    char& operator[](size_type pos);

    /*** substrings */
    string substr(size_type start, size_type n = npos); // the substring starting
                              // at position start containing n characters

    /*** inserting, replacing, and removing ***/
    string& insert(size_type pos, const string& s); // inserts s at position pos
    string& append(const string& s);                // appends s at the end
    string& replace(size_type start, size_type n, const string& s); // replaces n
                              // characters starting at pos with s
    void erase(size_type start = 0, size_type n = npos); // removes n
                              // characters starting at pos

    /*** assignment and concatenation ***/
    string& operator=(const string& s);  // also move assignment
    string& operator=(const char* cs);
    string& operator=(char ch);
    string& operator+=(const string& s); // also const char* and char

    /*** access to C-style string representation ***/
    const char* c_str();
    /*** finding things (see below) ***/
}
```

- Note that there is no constructor `string(char)`. Use `string(1, char)` instead.
- The subscript functions `operator[]` do not check for a valid index. There are similar `at()` functions that do check, and that throw `out_of_range` if the index is not valid.
- The `substr()` member function takes a starting position as its first argument and the number of characters as the second argument. This is different from the `substring()` method in `java.lang.String`, where the second argument is the end position of the substring.
- There are overloads of most of the functions. You can use C-style strings or characters as parameters instead of strings.
- Strings have iterators like library vectors.
- There is a bewildering variety of member functions for finding strings, C-style strings or characters. They all return `npos` if the search fails. The functions have the following signature (the `string` parameter may also be a C-style string or a character):

  ```
  size_type FIND_VARIANT(const string& s, size_type pos = 0) const;
  ```

  `s` is the string to search for, `pos` is the starting position. (The default value for `pos` is `npos`, not `0`, in the functions that search backwards).

  The "find variants" are `find` (find a string, forwards), `rfind` (find a string, backwards), `find_first_of` and `find_last_of` (find one of the characters in a string, forwards or backwards), `find_first_not_of` and `find_last_not_of` (find a character that is not one of the characters in a string, forwards or backwards).

  Example:

```
void f() {
        string s = "accdcde";
        auto i1 = s.find("cd");              // i1 = 2 (s[2]=='c' && s[3]=='d')
        auto i2 = s.rfind("cd");             // i2 = 4 (s[4]=='c' && s[5]=='d')
        auto i3 = s.find_first_of("cd");     // i3 = 1 (s[1]=='c')
        auto i4 = s.find_last_of("cd");      // i4 = 5 (s[5]=='d')
        auto i5 = s.find_first_not_of("cd"); // i5 = 0 (s[0]!='c' && s[0]!='d')
        auto i6 = s.find_last_not_of("cd");  // i6 = 6 (s[6]!='c' && s[6]!='d')
}
```

There are global overloaded operator functions for concatenation (`operator+`) and for comparison (`operator==`, `operator<`, etc.). They all have the expected meaning. Note that you cannot use `+` to concatenate a string with a number, only with another string, C-style string or character (this is unlike Java). In the new standard, there are functions that convert strings to numbers and vice versa: `stod("123.45") => double`, `to_string(123) => "123"`.

A1. Write a class that reads a file and removes HTML tags and translates HTML-encoded special characters. The class should be used like this:

```
int main() {
        TagRemover tr(std::cin); // read from cin
        tr.print(std::cout);     // print on cout
}
```

- All tags should be removed from the output. A tag starts with a < and ends with a >.
- You can assume that there are no nested tags.
- Tags may start on one line and end on another line.
- Line separators should be kept in the output.
- You don't have to handle all special characters, only `<`, `>`, ` `, `&` (corresponding to `< > space &`).
- Assignments like this should be a good fit for regular expressions. Study and use the C++ `regex` library if you're interested.

Copy the makefile from one of the previous labs, modify it, build and test.

A2. The Sieve of Eratosthenes is an ancient method for finding all prime numbers less than some fixed number M. It starts by enumerating all numbers in the interval $[0, M]$ and assuming they are all primes. The first two numbers, 0 and 1 are marked, as they are not primes. The algorithm then starts with the number 2, marks all subsequent multiples of 2 as composites, finds the next prime, marks all multiples, ... When the initial sequence is exhausted, the numbers not marked as composites are the primes in $[0, M]$.

In this assignment you shall use a string for the enumeration. Initialize a string of appropriate length to `PPPPP...PPP`. The characters at positions that are not prime numbers should be changed to `C`. Write a program that prints the prime numbers between 1 and 200 and also the largest prime that is less than 100,000.

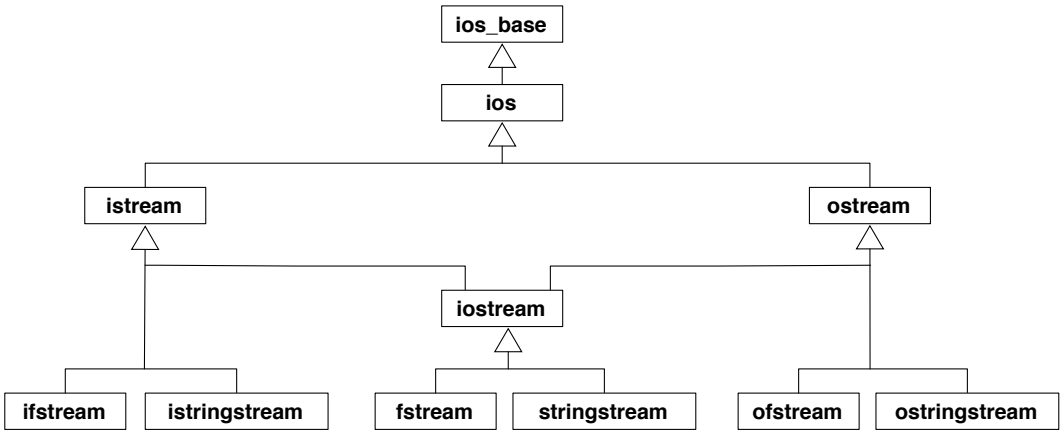
Example with the numbers 0–35:

```
                                  1         2         3
                        01234567890123456789012345678901234 5
Initial:                CCPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
Find 2, mark 4,6,8,...: CCPPCPCPCPCPCPCPCPCPCPCPCPCPCPCPCPCP
Find 3, mark 6,9,12,...: CCPPCPCPCCCPCPCCCPCPCCCPCPCCCPCPCCCP
Find 5, mark 10,15,20,...: CCPPCPCPCCCPCPCCCPCPCCCPCCCCCPCPCCCC
Find 7, mark 14,21,28,35: CCPPCPCPCCCPCPCCCPCPCCCPCCCCCPCPCCCC
Find 11, mark 22,33:    CCPPCPCPCCCPCPCCCPCPCCCPCCCCCPCPCCCC
...
```

3 The iostream Library

3.1 Input/Output of User-Defined Objects

In addition to the stream classes for input or output there are `iostream`'s that allow both reading and writing. The stream classes are organized in the following (simplified) hierarchy:



The classes `ios_base` and `ios` contain, among other things, information about the stream state. There are, for example, functions `bool good()` (the state is ok) and `bool eof()` (end-of-file has been reached). There is also a conversion operator `operator bool()` that returns true if the state is good, and a `bool operator!()` that returns true if the state is not good. We have used these operators with input files, writing for example `while (infile >> ch)` and `if (!infile)`.

To do formatted stream input and output of objects of user-defined classes, `operator>>` and `operator<<` must be overloaded.

A3. The files *date.h*, *date.cc*, and *date_test.cc* describe a simple date class. Implement the class and add operators for input and output of dates (`operator>>` and `operator<<`). Dates should be output in the form `2015-01-10`. The input operator should accept dates in the same format. (You may consider dates written like `2015-1-10` and `2015 -001 - 10` as legal, if you wish.)

The input operator should set the stream state appropriately, for example `is.setstate (ios_base::failbit)` when a format error is encountered.

3.2 String Streams

The string stream classes (`istringstream` and `ostringstream`) function as their "file" counterparts (`ifstream` and `ofstream`). The only difference is that characters are read from/written to a string instead of a file. In the following assignments you will use string streams to convert objects to and from a string representation (in the new standard, this can be performed with functions like `to_string` and `stod`, but only for numbers).

A4. In Java, the class `Object` defines a method `toString()` that is supposed to produce a "readable representation" of an object. This method can be overridden in subclasses.

Write a template function `toString` for the same purpose. Also write a test program. Example:

```
double d = 1.234;
Date today;
std::string sd = toString(d);
std::string st = toString(today);
```

You may assume that the argument object can be output with `<<`.

A5. Type casting in C++ can be performed with, for example, the `static_cast` operator. Casting from a `string` to a numeric value is not supported, since this involves executing code that converts a sequence of characters to a number.

Write a function template `string_cast` that can be used to cast a string to an object of another type. Examples of usage:

```
try {
        int i = string_cast<int>("123");
        double d = string_cast<double>("12.34");
        Date date = string_cast<Date>("2015-01-10");
} catch (std::invalid_argument& e) {
        cout << "Error: " << e.what() << endl;
}
```

You may assume that the argument object can be input with `>>`. The function should throw `std::invalid_argument` (defined in header `<stdexcept>`) if the string could not be converted.