Homework, C++ Programming

Class.

A.1. Implement a class Coding with two static methods:

```
/* For any character c, encode(c) is a character different from c */
static unsigned char encode(unsigned char c);

/* For any character c, decode(encode(c)) == c */
static unsigned char decode(unsigned char c);
```

Use a simple method for coding and decoding. Then write a program, encode, that reads a text file, encodes it, and writes the encoded text to another file. The command line:

```
./encode file
```

should run the program, encode *file*, and write the output to *file.enc*.

Write another program, decode, that reads an encoded file, decodes it, and writes the decoded text to another file *file.dec*. The command line should be similar to that of the encode program. Add rules to the makefile for building the programs.

Test your programs and check that a file that is first encoded and then decoded is identical to the original. Use the Unix `diff` command.

Note: the programs will work also for files that are UTF-8 encoded. In UTF-8 characters outside the “ASCII range” are encoded in two bytes, and the encode and decode functions will be called twice for each such character

Class string

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 imple- mentation 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.

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


    /*** finding things  ***/
    // see below

    /*** inserting, replacing, and  removing ***/
    void insert(size_type pos, const string& s); // inserts s  at  position  pos void
    append(const string& s);                          // appends s at the end
    void 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();
}
```

- 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 back- wards), **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. 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".

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
Initial:        CCPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
Find 2, mark 4,6,8,...: CCPPCPCPCPCPCPCPCPCPCPCPCPCPCPCPCP
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
```