CISS245: Advanced Programming
Assignment 3

Name: _____

Objectives

1. Compute with C-strings
2. Implement C-string functions
3. Write cpp and header file for a C-string library
4. Introduction to basic language processing

This assignment really leads to one program. Therefore once you're done with Q1, copy your Q1 folder to another folder for Q2 and continue work. Etc.

Remember that skeleton code, if given, need not be complete or correct. It is simply to give you something to start with. You should study the skeleton code carefully to understand it and correct/modify/improve it when necessary.

Skeleton

You will begin with the following skeleton files:

```
// File  : mystring.h
// Author: smaug

#ifndef MYSTRING_H
#define MYSTRING_H

int str_len(char []);
void str_cat(char [], char []);

#endif
```

```
// File  : mystring.cpp
// Author: smaug

#include "mystring.h"


int str_len(char x[])
{
    int len = 0;
    while (x[len] != '\0')
    {
        ++len;
    }

    return len;
}


void str_cat(char x[], char y[])
{
    int xlen = str_len(x);
    int ylen = str_len(y);

    for (int i = 0; i <= ylen; ++i)
    {
        x[xlen + i] = y[i];
    }
```

```
    return;
}
```

```
// File  : main.cpp
// Author: smaug

int main()
{
    return 0;
}
```

You will continually add code to the above for the first few questions as you develop your string library. (I.e, start with the above for Q1, after you're done, copy the files to Q2, etc.) The string library (`mystring.h` and `mystring.cpp`) is then used for the last question.

Stream I/O

The input processing done like this:

```
int x;
double y;
char z[1024];

std::cin >> x >> y >> z;
```

works fine. But note that for C-string input, whitespaces ends the input. For instance if you run this:

```
char z[1024];

std::cin >> z;
std::cout << z << std::endl;
```

and you attempt to enter two words as input into `z` you will see this:

```
hello world
hello
```

In other words `z` contains only `"hello"`. `world` is actually still in the "input stream" – it's not transferred to your program. You can of course get the second word `world` like this:

```
char z[1024];
char w[1024];

std::cin >> z >> w;
std::cout << z << std::endl;
std::cout << w << std::endl;
```

Then you will get this when you execute the program:

```
hello world
hello
world
```

What if you want to have the whole line of input into a C-string variable where the string contains whitespaces?

Run the following program:

```
#include <iostream>
#include <limits>

const int MAX_BUF = 1024;

int main()
{
    char s[MAX_BUF];
    std::cin.getline(s, MAX_BUF);
    std::cout << '[' << s << "]\n";

    return 0;
}
```

and enter a string with spaces.

If you want to writing a program that continually gets a string from the user, try the following program:

```
#include <iostream>
#include <limits>

const int MAX_BUF = 1024;

int main()
{
    while (1)
    {
        char s[MAX_BUF];
        std::cin.getline(s, MAX_BUF);
        if (std::cin.eof()) break;
        if (std::cin.fail() || std::cin.bad())
        {
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        }
        else
        {
            std::cout << '[' << s << "]\n";
        }
    }

    return 0;
}
```

Note in particular that your program must terminate when an EOF (i.e., end-of-file)

is reached:

```
if (std::cin.eof()) break;
```

On the MS windows console window, if you do Ctrl-Z and then press the enter key, you will get an EOF condition. On the Linux/Unix environment, you would do Ctrl-D.

To make your program robust and platform independent, you have to know that the string read from a stream might have non-visible characters at the end of the string. For instance on some systems, when a user press the enter key, the character `'\r'` is placed in the stream. The system might also placed `'\n'` in the string stream as well. This means that a string read from the stream (up to `'\n'`) might end with `'\r'` (and possibly also contain `'\n'`). You are strongly advised to remove all such right-trailing characters from your strings which are read from a stream. In other words after a string input process, say with the input going into character array variable `s`, you should look at the characters just before `'\0'` and see if there is `\r` or `\n` or both. For instance after an input, say `s` is `"hello world\r\n\0..."` you should make it `"hello world\0..."`

To analyze all the characters in your C-string `s`, you can of course print the ASCII code of the characters in `s` and have an ASCII table in front of you:

```
#include <iostream>
#include <limits>
#include "mystring.h"

const int MAX_BUF = 1024;

int main()
{
    while (1)
    {
        char s[MAX_BUF];
        std::cin.getline(s, MAX_BUF);
        if (std::cin.eof()) break;
        if (std::cin.fail() || std::cin.bad())
        {
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        }
        else
        {
            std::cout << '[' << s << "]\n";
            for (int i = 0; i < str_len(s); ++i)
            {
```

```
                std::cout << i << ' ' << s[i] << ' ' << int(s[i]) << std::endl;
            }
        }
    }
    return 0;
}
```

Some details about input

What happens during input? You should think of your `std::cin` as getting from a data stream, like an array of characters. When you type something on your keyboard, the characters are placed in the stream at the end. `std::cin` works by extracting characters from the front of the stream. (It's more like a queue.)

If you execute `std::cin.getline(s, 1024)`, `std::cin` will keep extracting characters until it sees a newline character. Character extracted from the input stream is placed in `s`. A `'\0'` is then added to `s`. Note that the newline character is extracted from the input stream but not placed in `s`. Note also that the `1024` in `std::cin.getline(s, 1024)` means that `std::cin` will only write at most 1023 characters into `s`, reserving one spot for `'\0'`. If `std::cin` reads more than 1023 characters, an error will occur.

A couple of things can go wrong ...

First, if `std::cin` reads more than 1023 characters without reading a newline character, you'll get an error.

Second, `std::cin` might read beyond the input stream, i.e., it has reached the EOF.

Third, something catastrophic might occur such as some hardware failure or something went wrong with the OS.

Fourth, instead of using `getline`, when you use `std::cin` to read an input for an integer, but what is read cannot be converted to an integer. For instance if you try to `std::cin >> i` where `i` is an `int`, but you enter `abc`.

`std::cin` maintains information on "eof", "fail", "bad" status using 3 bits (the eofbit, failbit, and badbit.) "eof" means you've reached the end of data. "bad" is the extreme case such as hardware or OS failure. "fail" is the other types mentioned above. So for instance if your program attempts to read data for an integer and the input is `"abc"`, then the fail bit is turn on, i.e., to 1. Once you've processed this error and you're ready to get the next input (if that's what you want to do), you need to set the fail bit to 0.

You use `std::cin.eof()`, `std::cin.fail()`, and `std::cin.bad()` to check if the eofbit, failbit, badbit is turned on. To set all these flags to 0, you do `std::cin.clear()`,

There are times when your want your `std::cin` to ignore some characters. For instance say you have input lines where each line has 3 data (say separated by spaces). If an error occurs, you might want your `std::cin` to throw away all input characters until the newline. In that case you do `std::cin.ignore(100, '\n')` which means

ignore at most 100 characters or until the newline character.

```
#include <iostream>
#include <limits>

const int MAX_BUF = 1024;

int main()
{
    while (1)
    {
        char s[MAX_BUF];
        std::cin.getline(s, MAX_BUF);
        if (std::cin.eof()) break;
        if (std::cin.fail() || std::cin.bad())
        {
            // CASE: Error in input
            // Do whatever can be done with s
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        }
        else
        {
            // CASE: No error in input
            // Do something with s
        }
    }

    return 0;
}
```

The above information on how inputs work applies to almost all input devices. Read the above and use the code from the previous section. Do NOT memorize the above information or the sample code from the previous section. The information is useful to know, but there's NO point in memorizing it. It's NOT computer science. In the future if you do work with low level I/O devices, just vaguely remember there are various forms of input failures. And if you need the C++ code for input just look for it here or online.

Q1. Write the string comparison function

```
int str_cmp(char x[], char y[]);
```

Add this prototype to `mystring.h` and implement the function in `mystring.cpp`.

The function returns 0 when the two strings `x` and `y` are the same, i.e., the characters of `x` up to `'\0'` matches exactly the characters of `y` up to `'\0'`. If they are different, then 1 is returned.

The file `main.cpp` should look like this:

a03q01/skel/main.cpp

```
#include <iostream>
#include <limits>
#include "mystring.h"

const int MAX_BUF = 1024;


void test_str_cmp()
{
    char s[MAX_BUF];
    char t[MAX_BUF];

    std::cin.getline(s, MAX_BUF);
    std::cin.getline(t, MAX_BUF);

    std::cout << str_cmp(s, t) << std::endl;
}


int main()
{
    int i = 0;
    std::cin >> i;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    switch (i)
    {
        case 0:
            test_str_cmp();
            break;
    }

    return 0;
}
```

Test 1.

```
0
abc
abc
0
```

Test 2.

```
0
 abc
abc
1
```

Test 3.

```
0
a bc
abc
1
```

Test 4.

```
0
hello world
hello world
0
```

You are strongly advised to try more test cases of your own.

Q2. Write the string copy function:

```
void str_cpy(char x[], char y[]);
```

The prototype should be added to `mystring.h` while the implementation of the function is in `mystring.cpp`.

The function copies `y` to `x`. For instance, if `y` is the string `"hello world"`, after calling `str_cpy(x, y)`, then `x` is `"hello world"`.

The following `main.cpp` must be included

a03q02/skel/main.cpp

```
#include <iostream>
#include <limits>
#include "mystring.h"

const int MAX_BUF = 1024;


void test_str_cmp()
{
    // earlier test function
}


void test_str_cpy()
{
    char x[MAX_BUF];
    char y[MAX_BUF];

    std::cin.getline(y, MAX_BUF);
    str_cpy(x, y);
    std::cout << x << std::endl;
    return;
}


int main()
{
    int i = 0;
    std::cin >> i;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    switch (i)
    {
        case 0:
            test_str_cmp();
            break;
```

```
        case 1:
            test_str_cpy();
            break;
    }
    return 0;
}
```

```
1
hello world
hello world
```

Test 5.

```
1
1 2 3
1 2 3
```

You are strongly advised to try more test cases of your own.

Q3. Write a "find character in string" function

```
int str_chr(char x[], char c);
```

that returns the index where `c` first occurs in `x`. If `c` does not occur in `x`, then -1 is returned.

For instance if `x` is `"hello world"` and `c` is `'o'`, then `str_chr(x, c)` returns 4.

a03q03/skel/main.cpp

```
#include <iostream>
#include <limits>
#include "mystring.h"

const int MAX_BUF = 1024;


void test_str_cmp()
{
    // earlier test function
}


void test_str_cpy()
{
    // earlier test function
}


void test_str_chr()
{
    char x[MAX_BUF];
    char c;

    std::cin.getline(x, MAX_BUF);
    std::cin >> c;

    std::cout << str_chr(x, c) << std::endl;
    return;
}


int main()
{
    int i = 0;
    std::cin >> i;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    switch (i)
```

```
    {
        // earlier cases
        case 2:
            test_str_chr();
            break;
    }
    return 0;
}
```

Test 1.

```
2
hello world
w
6
```

Test 2.

```
2
hello world???
?
11
```

You are strongly advised to try more test cases of your own.

Q4. Write a "find string in string" function

```
int str_str(char x[], char y[]);
```

that returns the index where `y` first occurs in `x`. If `y` does not occur in `x`, then -1 is returned.

For instance if `x` is `"hello world"` and `y` is `" wor"`, then `str_str(x, y)` returns 5.

a03q04/skel/main.cpp

```
#include <iostream>
#include <limits>
#include "mystring.h"

const int MAX_BUF = 1024;


// earlier test functions


void test_str_str()
{
    char x[MAX_BUF];
    char y[MAX_BUF];

    std::cin.getline(x, MAX_BUF);
    std::cin.getline(y, MAX_BUF);

    std::cout << str_str(x, y) << std::endl;
    return;
}


int main()
{
    int i = 0;
    std::cin >> i;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    switch (i)
    {
        // earlier cases
        case 3:
            test_str_str();
            break;
    }
    return 0;
}
```

Test 1.

```
3
hello world
 wor
5
```

Test 2.

```
3
hello world
or
7
```

Test 3.

```
3
hello world
wor
-1
```

Test 4.

```
3
abc def defg defghi
defgh
13
```

You are strongly advised to try more test cases of your own.

Q5. Write a lowercase function:

```
void str_lower(char x[], char y[]);
```

that copies the character of the string `y` to `x` except that uppercase characters are replaced by lowercase.

For instance if `y` is `"HelLo worlD ... 123!"`, then after calling `str_lower(x, y)`, `x` is
`"hello world ... 123!"`.

a03q05/skel/main.cpp

```
#include <iostream>
#include <limits>
#include "mystring.h"

const int MAX_BUF = 1024;


// earlier test functions


void test_str_lower()
{
    char x[MAX_BUF];
    char y[MAX_BUF];

    std::cin.getline(y, MAX_BUF);
    str_lower(x, y);

    std::cout << x << std::endl;
    return;
}


int main()
{
    int i = 0;
    std::cin >> i;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    switch (i)
    {
        // earlier cases
        case 5:
            test_str_lower();
            break;
    }
    return 0;
```

```
}
```

Test 1.

```
5
hEllo woRld
hello world
```

Test 2.

```
5
1 2 3 4 5
1 2 3 4 5
```

You are strongly advised to try more test cases of your own.

Q6. Write a string "tokenizing" function:

```
bool str_tok(char x[], char y[], char delimiters[]);
```

that does the following.

If `y` is `"hello world"` and `delimiters` is `" "`, when after calling `str_tok(x, y, delimiters)`, `x` becomes `"hello"`, `y` becomes `"world"`. Furthermore `true` is returned. Basically the characters in `delimiters` is used to cut up the string `y` (once). At this point, if `str_tok(x, y, delimiters)` is called again, `x` becomes `"world"`, `y` becomes `""`, and the function return `true`. If we call `str_tok(x, y, delimiters)` a third time, `x` becomes `""`, `y` stays as `""`, and `false` is returned.

Note that `delimiters` can contain more than one characters. For instance, suppose `y` is
`"hello world,galaxy,universe!"`. If `delimiters` is `" ,"`, then after the first call of `str_tok(x, y, delimiters)`, `x` becomes `"hello"`, `y` becomes `"world,galaxy,universe!"`, and the function return `true`. After the second call of `str_tok(x, y, delimiters)`, `x` becomes `"world"`, `y` becomes `"galaxy,universe!"`, and the function return `true`. After the third call of `str_tok(x, y, delimiters)`, `x` becomes `"galaxy"`, `y` becomes `"universe!"`, and the function return `true`. After the fourth call of `str_tok(x, y, delimiters)`, `x` becomes `"universe!"`, `y` becomes `""`, and the function return `true`. After the fifth call of `str_tok(x, y, delimiters)`, `x` becomes `""`, `y` stays as `""`, and the function return `false`. In this example, `' '` and `','` are used to cut up `y`.

Note that if `y` is `",123"` and `delimiters` is `","`, then on calling `str_tok(x, y, delimiters)`, `x` is `""`, `y` is `"123"`, and `true` is returned.

Note also that because there are multiple characters in delimiters, it's possible for string `y` to be cut up by the delimiter characters in different ways. The character that is used is the one that produces the shortest `x`. For instance in the above case where `y` is
`"hello world,galaxy,universe!"` and `delimiters` is `" ,"`. `y` can be cut up by `' '` (at index 5), by `','` (at index 11), and by `' '` (at index 18). The delimiter character that is actually used is `' '` (at index 5) because that creates the shortest left substring `x`.

Make sure you try some of your own test cases.

`a03q06/skel/main.cpp`

```
#include <iostream>
#include <limits>
#include "mystring.h"
```

```
const int MAX_BUF = 1024;


// earlier test functions


void test_str_tok()
{
    char x[MAX_BUF];
    char y[MAX_BUF];
    char delimiters[MAX_BUF] ~textred!= " ,."@;

    std::cin.getline(y, MAX_BUF);
    bool b = str_tok(x, y, delimiters);

    std::cout << b << ' ' << x << ' ' << y << std::endl;
    return;
}


int main()
{
    int i = 0;
    std::cin >> i;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    switch (i)
    {
        // earlier cases
        case 6:
            test_str_tok();
            break;
    }
    return 0;
}
```

Q7. Write a chat bot. The chat bot will learn to recognize the user. Here's a test run:

```
Hi, what is your name?
My name is John.
Hi John. How are you?
```

The chat bot must recognize the name of the user when given responses of the following form:

- `My name is John.`
- `I am John.`
- `I'm John.`
- `John.`

and also when the user enters extraneous spaces such as

- `My  name  is  John.`
- `I  am  John.`
- `I'm  John.`
- `John.`

and also when the user forgot to enter the period (`'.'`) and when the user forgets to capitalize correctly such as

- `my  name  is  john.`
- `i  am  john.`
- `i'm  john.`
- `john.`

It's helpful to write a `str_capitalize()` function such that `str_capitalize(x)` will replace `x[0]` with its uppercase. It's also useful to have a function that strips away left trailing space, `str_lstrip` (the left strip function) so that if you execute `str_lstrip(x)` where `x` is `"  hello world  "`, `x` becomes `"hello world  "`.

It's also convenient to have a corresponding `str_rstrip` (the right strip function). If you execute `str_rstrip(x)` where `x` is `"  hello world  "`, `x` becomes `"  hello world"`.

With the left and right strip functions, it's easy to create `str_strip` (the strip function) which performs both the left and right strip. If you execute `str_strip(x)` where `x` is `"  hello world  "`, `x` becomes `"hello world"`.

Q8. The next thing that the chat bot does is to be helpful to the user:

```
Hi, what is your name?
My name is John.
Hi John. How are you?
I'm sad.
I'm sorry to hear that. Why are you sad?
```

Again, the chat bot must recognize the following variations:

- `I'm sad.`
- `i'm sad.`
- `I am sad.`
- `i am sad.`
- `Sad.`
- `sad`

as well as without period and with extraneous spaces. Your chat bot must also recognize `depressed` and `miserable` (you can add other synonyms if you like). So here's another chat session:

```
Hi, what is your name?
John
Hi John. How are you?
I'm depressed.
I'm sorry to hear that. Why are you depressed?
```

The chat bot must also respond to the following emotions: `happy`, `glad`. (You can add others if you like.) Here's another chat session:

```
Hi, what is your name?
My name is John.
Hi John. How are you?
I'm happy.
I'm glad to hear that. Why are you happy?
```

Your chat bot must also recognize words like `very`, `extremely`, and `really`.

```
Hi, what is your name?
My name is John.
Hi John. How are you?
I'm very sad.
I'm sorry to hear that. Why are you so sad?
```

(Note the new word `so` when the user adds `very`.)

Once the user respond, the program does this:

```
Hi, what is your name?
My name is John.
Hi John. How are you?
I'm very sad.
I'm sorry to hear that. Why are you so sad?
My dog died.
I'm sorry. Tell me about your dog.
```

Here's another execution:

```
Hi, what is your name?
My name is John.
Hi John. How are you?
I'm very sad.
I'm sorry to hear that. Why are you so sad?
My goldfish died.
I'm sorry. Tell me about your goldfish.
```

Your program should respond to a general statement like:

```
Hi, what is your name?
My name is John.
Hi John. How are you?
I'm very sad.
I'm sorry to hear that. Why are you red sad?
My [pet/friend/relative/etc.] died.
I'm sorry. Tell me about your [pet/friend/relative/etc.].
```

On the happy side, your program should do this: You program should respond to a general statement like:

```
Hi, what is your name?
My name is John.
Hi John. How are you?
I'm very happy.
I'm glad to hear that. Why are you so happy?
I got a raise.
I'm glad. Tell me about your raise.
```

or

```
Hi, what is your name?
My name is John.
Hi John. How are you?
I'm very happy.
I'm glad to hear that. Why are you so happy?
I bought a car.
I'm glad. Tell me about your car.
```

You only need to handle `got`, `bought`, and `received`.

Q9. The last thing to add is a certain amount of randomness. Instead of always

```
Hi, what is your name?
```

vary it randomly with

```
Hello, what is your name?
```

(Of course use the random number generator). And instead of

```
I'm sorry to hear that. Why are you sad?
```

or

```
I'm glad to hear that. Why are you happy?
```

vary it randomly with

```
I'm so sorry to hear that. Why are you sad?
```

or

```
I'm so glad to hear that. Why are you happy?
```

and

```
I'm sorry to hear that. Tell me why you are sad.
```

or

```
I'm glad to hear that. Tell me why you are happy.
```

You are of course encouraged to write more string functions to clean up your main program.

This simple chat bot is just an illustration of what you can do to simulate human behavior – and of course it is also a practice on string processing. To create a strong AI bot requires a lot more work. Obviously you need to know lots of CS theory, algorithms, especially algorithms related to language theory and linguistics, and especially AI.