HW1

- Due: Tues. Sep. 4, 2018, 11:59pm (PST)
- Directory name in your github repository for this homework (case sensitive): `hw1`
    - Once you have cloned your `hw-usc_username` repo, create this `hw1` folder underneath it (i.e. `hw-usc_username/hw1`).
    - If your `hw-usc_username` repo has not been created yet, please do your work in a separate folder and you can copy over relevant files before submitting.

Objective

In this assignment we will review course polices, using GitHub, and resources such as skeleton code. We will also review streams and recursion. We will explore pointers and memory management by writing simple character storage managers ourselves. In doing so, we will not only improve our practice of pointers, but also develop greater appreciation for what the C++ language and the operating system provide for us in memory management. We will also practicing using be dynamic memory allocation by using dynamically allocated arrays.

A Few Notes on Repositories

1. Never clone one repo into another. If you have a folder `cs104` on your VM and you clone your personal repo `hw-usc_username` under it (i.e., `cs104/hw-usc_username`) then whenever you want to clone some other repo (such as the repo we use to give you skeleton code, or another personal repo you may have created), you need to do it back up in the `cs104` folder

folder.
2. Your repos may not be ready immediately but I to create your GitHub account and fill out the GitHub information form linked to at the end of Lab 01.

Skeleton Code

On many occasions we will want to distribute skeleton code, tests, and other pertinent files. To do this, we have made a separate repository, `homework-resources`, under our class GitHub site. You should clone this repository to your laptop and do a `git pull` regularly to check for updates. (Yes, even we sometimes make mistakes; when we do, we will fix them as quickly as possible, but you'll only get the fixes when you pull.)

```
$ git clone git@github.com:usc-csci104-fall2018/homework-r
```

Again, be sure you don't clone this repo into your `hw-usc_username` repo, but at some higher-up point like in a `cs104` folder on your laptop.

Note: If you can't access the repository, you can find the files here: palindrome, first_memtest, simplecharmanagercpp, simplecharmanagerh, second_memtest, flexcharmanagercpp, and flexcharmanagerh.

Problem 1 (Course Policies, 10%)

Carefully study the information on the course web site, then answer the following questions about course policies (anywhere from no to all answers may be correct):

Place your answers to this question in a file named `hw1.txt`

Part (a):

Which of the following are acceptable behaviors in solving homeworks/projects?

1. Looking up information relevant to the course online.
2. Looking up or asking for sample solutions online.
3. Copying code from my classmates, and then editing it significantly.
4. Asking the course staff for help.

5. Sitting next to my classmate and coding togeth
   team or with significant conversation about our
   approach.
6. Sharing my code with a classmate, if he/she just promises not to copy them, but to just read over it and learn from it.
7. Copying graded test cases from my classmate and then editing them.
8. Copying test cases from my classmates or Piazza when test cases are not graded.

Part (b):

Which of the following are recommended ways of writing code?

1. gedit
2. emacs
3. Eclipse
4. sublime
5. Microsoft Visual Studio
6. notepad

Part(c):

What is the late submission policy?

1. Each assignment can be submitted late for 50% credit.
2. Each student has 3 late days of which only one can be used per HW.
3. Students need to get approval before submitting an assignment late.
4. There is no late policy.

Part(d):

After making a late submission by pushing your code to Gihub you should...

1. Do nothing. Sit back and enjoy.
2. Complete the normal homework submission form.
3. Email your instructor.
4. Start the next assignment sooner.

Problem 2 (Git, 10%)

Carefully review and implement the steps discussed in Lab1. Then answer the following questions:

Continue your answers to this question in the file named `hw1.txt`

Part (a):

Which of the following git user interfaces are accepted and supported in this course?

1. Git Bash (Windows)
2. GitHub Desktop Client
3. Terminal (Mac or Linux)
4. Eclipse eGit
5. Tower Git Client

Part (b):

Provide the appropriate git command to perform the following operations:

1. Stage an untracked file to be committed. The (hypothetical) file is called 'hw1q2b.cpp'.
2. Display the details of the last three commits in the repository.

Part (c)

Let's say you staged three files to be committed. Then, you ran the following command:

```
git commit
```

What will git do?

Problem 3 (Review Material and Programming Advice)

Carefully review recursion and dynamic memory management from your CSCI 103 notes and textbook. You may also find Chapters 2 and 5 from the textbook, Chapters 2 and 3 from the lecture notes, and the C++ Interlude 2, helpful.

You will lose points if you have memory leaks, so be sure to run `valgrind` once you think your code is working.

```
$ valgrind --tool=memcheck --leak-check=yes ./program input.txt output.txt
```

Hint: In order to read parameters as command line arguments in C++, you need to use a slightly different syntax for your `main` function: `int main (int argc, char *`

`argv[])`. When your program is called at the comma
`argc` will then contain the total number of argument
the program was given, and `argv` will be an array of
the parameters the program was passed. `argv[0]` is always the name of your program, and `argv[1]` is the first argument. The operating system will assign the values of `argc` and `argv`, and you can just access them inside your program.

Problem 4 (Recursion and Streams, 10%)

Remember that a string being a palindrome can be characterized as follows:

- The empty string is a palindrome.
- Any single character is a palindrome.
- If `c` is a character, and `p` is a palindrome, then the string `cpc` is also a palindrome.

Consider the program `palindrome.cpp`, which we have provided for you in the `homework_resources/hw1` folder/repo. Copy the file to your `hw_usc-username/hw1` folder. The program is intended to receive a string as input, and output "Palindrome" indicating that the string is a palindrome, or "Not a Palindrome" indicating that it isn't. However, the program currently outputs "Palindrome" for all strings. Copy, compile, and run the program with several different inputs to verify this.

Your job is to fix the code so that it gives the correct output for all possible inputs. Only make changes where the code indicates that they should be made: you should not change the main function, nor the start of the helper function. Your program file should be named `palindrome.cpp`.

Problem 5 (Streams, 15%)

Write a program that reads in a text file and outputs to cout the words of that file in reverse order, one per line. To achieve this, you will either want to use recursion or dynamic memory allocation (your choice). The program should receive the input filename as a command line parameter.

For the text file, the first line will consist only of an integer representing the number of words in the file; let's call it `n`. This will be followed by `n` words. Each word will consist

only of letters (uppercase or lowercase), and the wor
be separated by some amount of whitespace (space
newline).

The following is a sample input and output of the program.

- Input text file: hw1q5.txt

```
11
Hello World
saying this like Yoda
But soylent green are people
```

- Compiling the program

```
> g++ -g hw1q5.cpp -o hw1q5
```

- Running the program and the output

```
> ./hw1q5 hw1q5.txt
people
are
green
soylent
But
Yoda
like
this
saying
World
Hello
```

Your program file should be named `hw1q5.cpp`.

Problem 6 (Memory)

In class you learned about the memory stack that C++ uses to store variables. In this problem you will replicate this functionality by creating your own memory manager, which a user will be able to store and lookup variables in.

Part 1: A Simple, Contiguous Character Storage Manager (20%)

The memory will be stored in a single array of `10000` characters

```
/* char buffer[10000] */
```

You will need to implement the following two functions:

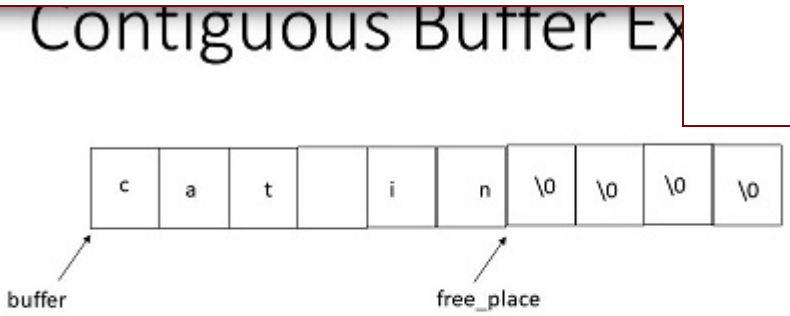
1. `char* alloc_chars( int n);` This function will return a pointer to a memory that can hold n characters. If there is not enough space left in the buffer, return null.
2. `void free_chars(char *);` This function given a pointer in the buffer will free all memory addressed from that address in the buffer until the end of the buffer. This is not the ideal way to free memory for obvious reasons, but it is a LOT easier.

So, someone who is using your program might utilize it by writing the following code:

```
char* c1 = alloc_chars(2);
c1[0] = 'h';
c1[1] = 'i';
char* c2 = alloc_chars(3);
cout << c1[0] << c1[1] << endl;
free_chars(c1);
```

Skeleton code is given in the `simpleCharManager.*` and `first_memtest.cpp`. If a character has not been allocated to the user, then it should contain the null character. So, all characters should be null before anything has been allocated, and characters should be null after they have been freed.

In this part of the problem, all characters are stored contiguously in memory. When memory is requested by your program, you will assign it the smallest available array index, unless there is not enough remaining space in the buffer. If memory is freed, it will be freed from the address requested until the end of the buffer. This is illustrated below:



This figure demonstrates how memory must be contiguous in t simplest character manager buffer. Buffer indicates the start of buffer and free_place is a pointer to the next available characte buffer. All of the free memory in the character buffer contains t character, '\0'.

After you have your memory manager, you'll need to do the following tasks:

1. Write a small program that utilizes your memory manager. Your program should store the phrase: `Hello world!\n` one character at a time, storing whitespace and punctuation also. Then print the phrase, then release the memory for `world!\n` and finally obtain memory to store the phrase `moon! Bye.\n`. You may only use your memory manager to do this. You may not use `malloc`, `new`, `free`, `delete`, or C++ strings.
2. Draw what the stack memory of the driver program looks like, and submit that as `stack.xyz` (this can be stack.jpg, or stack.txt, or whatever format you produce it in).
3. This storage manager is very problematic. Explain how it is possible to overwrite part of the buffer that the user still has a pointer to (and may try to access via that pointer). Write these answers in `hw1.txt`.

Part 2: A More Flexible Character Storage Manager (35%)

Being unable to selectively remove a few characters in the middle of our program stack is not ideal. To fix the identified problems, we will need to keep careful track of each of our pointers.

We will again use a large (10000) buffer of characters that we will directly manage. Once again, if the character is not allocated, then it should hold the null character.

Since we will now allow the user to delete characters in the middle of our buffer, while leaving the characters after, we will need to keep careful track of which characters are allocated (we cannot merely keep track of the last character in use). In addition, we will need to keep careful track of how many chars are assigned via a specific `alloc_chars` function call. When you call `alloc_chars`, you will need to create a `Mem_Block`, which is defined in the skeleton code, which will keep track of both the `char*` return value, as well as the number of characters stored in this memory block. You should store all of your `Mem_Block`s in a dynamically allocated array. How big should your array be?

- Start with an array of size `2`.
- If you run out of space, then double the size of the array (make sure to not have memory leaks!)
- Unless your array is size `2`, then the allocated elements in your array should be no more than 4 times the number of active memory blocks in use.
- If you have too many allocated elements, then you should halve the size of your array (make sure to not have memory leaks!)

In case you are confused, keep in mind that there are two different arrays here. You have your original array of `10000` characters which is statically allocated. Then you have a dynamically allocated array storing your `Mem_Block` structs, which will vary in size according to the above specifications.

Skeleton code for this part is provided in `flexCharManager.*` and `second_memtest.cpp`. You will need to implement the same functions as in the first part, but now the memory can be assigned from any place in your buffer as long as you have space. As a consequence we will also be able delete individual pointers from memory.

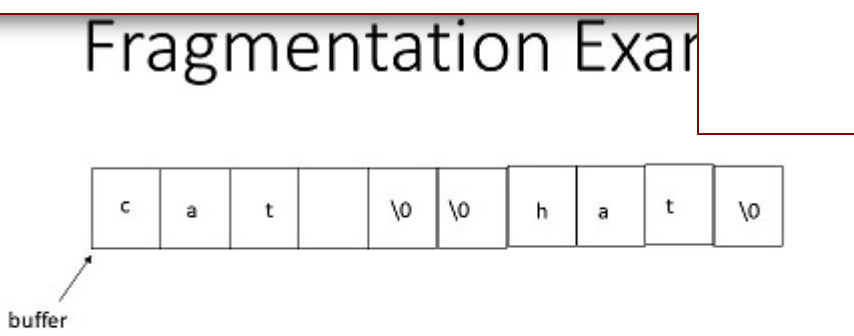
Implement the same two functions as before:

1. `char * alloc_chars( int n);` This function will return a pointer to memory that can hold `n` characters. For determining from where to allocate memory, implement a first fit strategy. Starting from the beginning of the character buffer, examine the buffer for the first space available that can satisfy the entire request and return the address to that. The address returned should be the address closest to the starting

address of the buffer that can fit the entire n re
characters. If there is no space left in the buffe
can satisfy the request, return null. You will hav
keep track of the amount of memory that was
allocated in order to be able to free it later.
2. `void free_chars(char *);` This function given a pointer in the buffer will free all memory at the given address that had been allocated in an `alloc_chars` call. It will not free all memory left in the buffer, only the memory allocated at the given address from a single `alloc_char` call. For example, if a call `alloc_chars(10)` was made and returned `address_1`, then a call to `free_chars(address_1)` would release `10` characters from buffer starting at `address_1`, overwriting the contents with the null character. If the pointer given as an argument to `free_chars` is not a valid address in the buffer range or has not been requested from `alloc_chars`, then nothing will be done. In order to know what memory addresses are valid, you must manage the memory blocks in use.

Some additional notes:

1. Your character storage manage must have the exact same behavior as any sequence of `new char[int]` and `delete` calls (with the exception of running out of memory faster).
2. There is one major problem with this implementation. We may have enough space in the buffer, but it is not all contiguous, so when a request for more memory comes, the implementation will not find enough space. This is called fragmentation and it is illustrated below:



This figure demonstrates how memory fragmentation can occur when memc allocated or removed from any place in the buffer. First there was a request f in which the string "cat " is placed. Then a request for 2 chars is made in whic string "in" is placed. Then a request for 3 chars is made in which the string "h placed. Then the 2 chars requested for "in" are freed. When a request for 3 c made in which to place the word "top", it will fail because even though 3 cha available in the buffer, they are not 3 contiguous chars in the buffer.

Given the following requests for memory and free memory and words placed in the space, diagram what the fragmentation looks like in a buffer of 20 characters assuming that a first fit algorithm is used to find space in the buffer, and submit your work as file `fragmentation.xyz`.

```
Request 3 chars in which to place the string "in "
Request 7 chars in which to place the string "French "
Request 7 chars in which to place the string "chapeau"
Remove the 3 chars allocated for "in "
Request 3 chars in which to place the string "top"
Remove the 7 chars allocated for "French "
Request 8 chars in which to place the string "sombrero"
Request 3 chars in which to place the string "hat"
```

Completion Checklist

- Directory name for this homework (case sensitive): `hw1`
  - This directory should be in your `hw-username` repository
  - This directory needs its own `README.md` file briefly describing your work
  - Your completed `hw1.txt`
  - Your completed `palindrome.cpp` implementation
  - Your completed `hw1q5.cpp` implementation
  - Your completed `stack.xyz` file
  - Your completed `fragmentation.xyz` file

- Your completed simple character manag
  driver in the following files:
  `simpleCharManager.h`, `simpleCharManag`
  and `first_memtest.cpp`
- Your completed flexible character manager and driver in the following files: `flexCharManager.h`, `flexCharManager.cpp` and `second_memtest.cpp`

Commit then Re-clone your Repository

You can submit your homework here. Please make sure you have read and understood the submission instructions.

WAIT! You aren't done yet. Complete the last section below to ensure you've committed all your code.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your directory to your `hw-usc-username` repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. Go to your home directory: `$ cd ~`
2. Create a `verify` directory: `$ mkdir verify`
3. Go into that directory: `$ cd verify`
4. Clone your hw-username repo: `$ git clone git@github.com:usc-csci104-fall2018/hw-username.git`
5. Go into your folder `$ cd hw-username/`
6. Recompile and rerun your programs and tests to ensure that what you submitted works.