CIS2107_Lab09: String Library

Points: **100** points

Objective:

In this assignment, you'll create your own library of string functions. You'll have the opportunity to practice manipulating strings and managing memory. Additionally, you'll learn the role of header and library files.

You may **not** call functions in `string.h` but you can use other code in the Standard C Library.

Task:

Functions to Include in the Library

Implement each of the following functions. Be sure that any string that you create or modify is in fact a string, *i.e.*, an array of char terminated with the null character, '`\0`'.

Additionally, you should write a driver which tests each of these functions on real data.

- **[2 points]** `int all_letters(char *s)`

  Returns `1` if all of the characters in the string are either upper- or lower-case letters of the alphabet. It returns `0` otherwise

- **[2 points]** `num_in_range(char *s1, char b, char t)`

  returns the number of characters `c` in `s1` such that `b<=c<=t`

- **[4 points]** `diff(char *s1, char *s2)`

  returns the number of positions in which `s1` and `s2` differ, i.e., it returns the number of changes that would need to be made in order to transform `s1` into `s2`, where a change could be a character substitution, an insertion, or a deletion.

- **[2 points]** `void shorten(char *s, int new_len)`

  Shortens the string s to `new_len`. If the original length of `s` is less than or equal to `new_len`, `s` is unchanged

- **[2 points]** `int len_diff(char *s1, char *s2)`

  Returns the length of `s1` - the length of `s2`

- **[2 points]** `void rm_left_space(char *s)`

  removes whitespace characters from the beginning of `s`

- **[2 points]** `void rm_right_space(char *s)`

  removes whitespace characters from the end of `s`

- **[2 points]** `void rm_space(char *s)`

  removes whitespace characters from the beginning and the ending `s`

- **[5 points]** `int find(char *h, char *n)`

  returns the index of the first occurrence of `n` in the string `h` or `-1` if it isn't found.

- **[5 points]** `char *ptr_to(char *h, char *n)`

  returns a pointer to the first occurrence of n in the string `h` or `NULL` if it isn't found

- **[4 points]** `is_empty(char *s)`

  returns `1` if `s` is `NULL`, consists of only the null character ('') or only whitespace. returns `0` otherwise.

- **[5 points]** `str_zip(char *s1, char *s2)`

  Returns a new string consisting of all of the characters of `s1` and `s2` interleaved with each other. For example, if `s1` is "`Spongebob`" and `s2` is "`Patrick`", the function returns the string "`SPpaotnrgiecbkob`"

- **[5 points]** `void capitalize(char *s)`

  Changes `s` so that the first letter of every word is in upper case and each additional letter is in lower case.

- **[5 points]** `int strcmp_ign_case(char *s1, char *s2)`

  Compares `s1` and `s2` ignoring case. Returns a positive number if `s1` would appear after `s2` in the dictionary, a negative number if it would appear before `s2`, or `0` if the two are equal.

- **[3 points]** `void take_last(char *s, int n)`

  Modifies `s` so that it consists of only its last `n` characters. If `n` is ≥ the length of `s`, the original string is unmodified. For example if we call take_last("Brubeck" 5), when the function finishes, the original string becomes "ubeck"

- **[5 points]** `dedup(char *s)`

  returns a new string based on `s`, but without any duplicate characters. For example, if `s` is the string, `"There's always money in the banana stand."`, the function returns the string `"Ther's alwymonitbd."`. It is up to the caller to free the memory allocated by the function.

- **[5 points]** `pad(char *s, int d)`

  returns a new string consisting of all of the letters of `s`, but padded with spaces at the end so that the total length of the returned string is an even multiple of `d`. If the length of `s` is already an even multiple of `d`, the function returns a copy of `s`. The function returns `NULL` on failure or if s is `NULL`. Otherwise, it returns the new string. It is up to the caller to free any memory allocated by the function.

- **[5 points]** `ends_with_ignore_case(char *s, char *suff)`

  returns `1` if `suff` is a suffix of `s` ignoring case or `0` otherwise.

- **[5 points]** `char *repeat(char *s, int x, char sep)`

  Returns a new string consisting of the characters in `s` repeated `x` times, with the character `sep` in between. For example, if `s` is the string `all right`, `x` is 3, and `sep` is `,` the function returns the new string `all right,all right,all right`. If s is `NULL`, the function returns `NULL`. It is up to the caller to free any memory allocated by the function.

- **[5 points]** `char *replace(char *s, char *pat, char *rep)`

  Returns a copy of the string s, but with each instance of pat replaced with rep, note that `len(pat)` can be less than, greater than, or equal to `len(rep)`. The function allocates memory for the resulting string, and it is up to the caller to free it. For example, if we call `replace("NBA X", "X", "rocks")`, what is returned is the new string `NBA rocks` (but remember, `pat` could be longer than an individual character and could occur multiple times).

- **[5 points]** `char *str_connect(char **strs, int n, char c)`

  Returns a string consisting of the first `n` strings in `strs` with the character `c` used as a separator. For example, if `strs` contains the strings `{"Washington", "Adams", "Jefferson"}` and `c` is `'+'`, the function returns the string `"Washington+Adams+Jefferson"`

- **[5 points]** `void rm_empties(char **words)`

  `words` is an array of string terminated with a `NULL` pointer. The function removes any empty strings (*i.e.*, strings of length `0`) from the array.

- **[5 points]** `char **str_chop_all(char *s, char c)`

  Returns an array of string consisting of the characters in `s` split into tokens based on the delimeter `c`, followed by a `NULL` pointer. For example, if `s` is "I am ready for a nice vacation" and `c` is ' ', it returns `{"I", "am", "ready", "for", "a", "nice", "vacation", NULL}`

Pointer vs Array Notation

Though it's not a formal requirement, it is suggested that you try to do some of these using pointer notation instead of array notation.

For example, we could write a string length function as:

```
int strlen(char s[])
{
   int i=0;
   while (s[i]!='\0')
      i++;
   return i;
}
```

We could also write:

```
int strlen(char *s)
{
   char *t=s;

   while (*t!='\0')
      t++;
   return t-s;
}
```

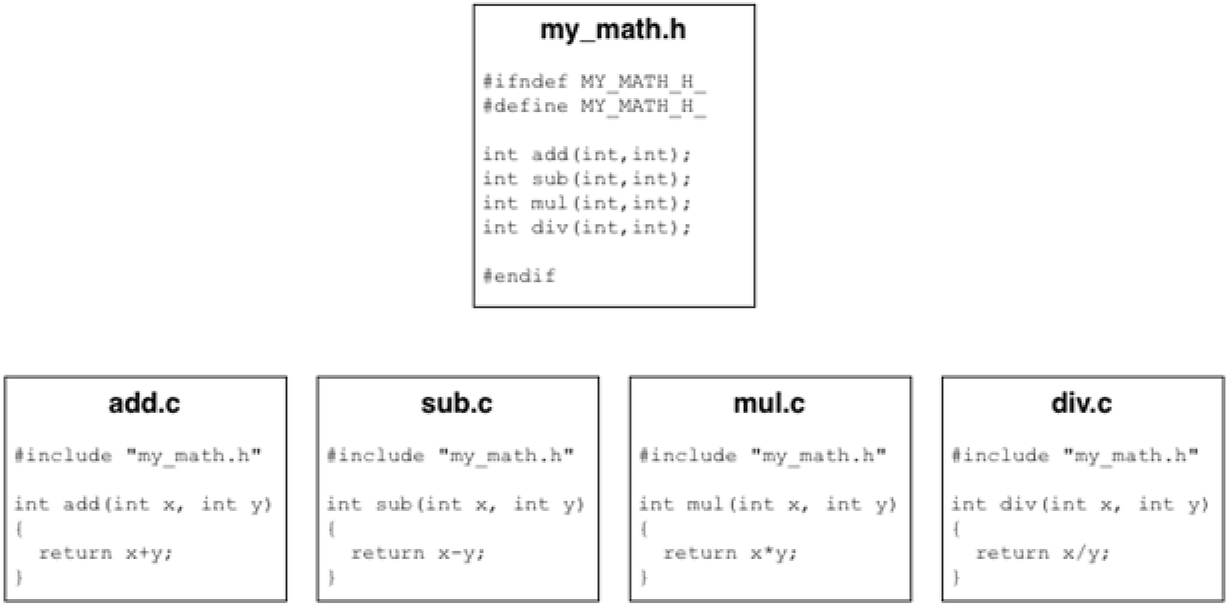
Arch

Remember that we keep our function declarations in a header file, a `.h` file. Each of your functions should be in a separate `.c` file (e.g., you'll have `all_letters` in a file called `all_letters.c` file, the `num_in_range` in a file called `num_in_range.c`.

So you'll have:

- A collection of `.c` files:
  - one for each function in the library
  - a test program
- A single `.h` file, which includes the declarations for each of the functions in the library. This is `#included` by each of the `.c` files.

This is just as we did with our little practice math library from class. Recall that the organization was:



my_math.h

```
#ifndef MY_MATH_H_
#define MY_MATH_H_

int add(int,int);
int sub(int,int);
int mul(int,int);
int div(int,int);

#endif
```

add.c

```
#include "my_math.h"

int add(int x, int y)
{
  return x+y;
}
```

sub.c

```
#include "my_math.h"

int sub(int x, int y)
{
  return x-y;
}
```

mul.c

```
#include "my_math.h"

int mul(int x, int y)
{
  return x*y;
}
```

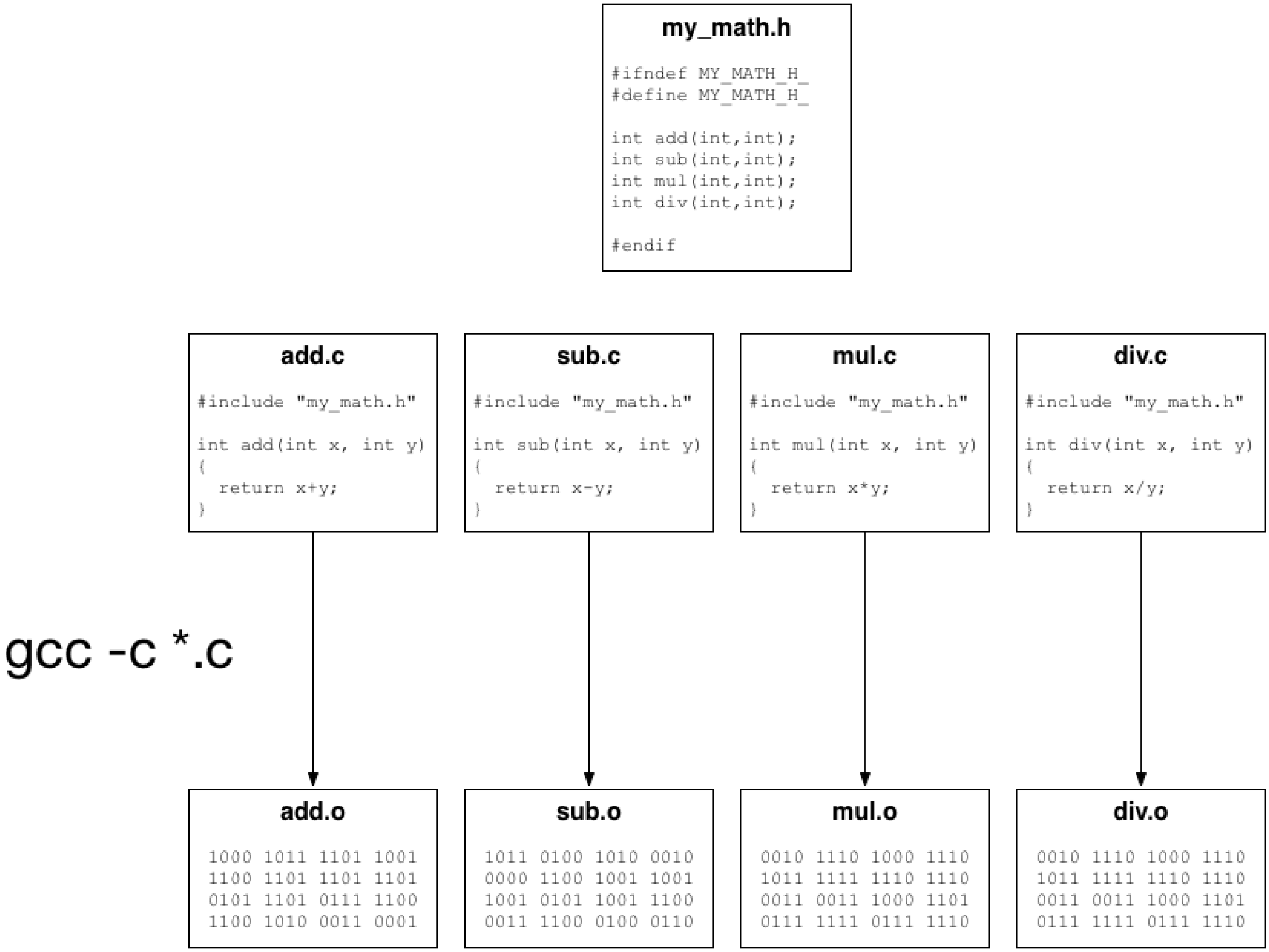
div.c

```
#include "my_math.h"

int div(int x, int y)
{
  return x/y;
}
```

We then compiled the `.c` files (without linking) by typing

```
gcc -c *.c
```



my_math.h

```
#ifndef MY_MATH_H_
#define MY_MATH_H_

int add(int,int);
int sub(int,int);
int mul(int,int);
int div(int,int);

#endif
```

add.c

```
#include "my_math.h"

int add(int x, int y)
{
  return x+y;
}
```

sub.c

```
#include "my_math.h"

int sub(int x, int y)
{
  return x-y;
}
```

mul.c

```
#include "my_math.h"

int mul(int x, int y)
{
  return x*y;
}
```

div.c

```
#include "my_math.h"

int div(int x, int y)
{
  return x/y;
}
```

add.o

```
1000 1011 1101 1001
1100 1101 1101 1101
0101 1101 0111 1100
1100 1010 0011 0001
```

sub.o

```
1011 0100 1010 0010
0000 1100 1001 1001
1001 0101 1001 1100
0011 1100 0100 0110
```

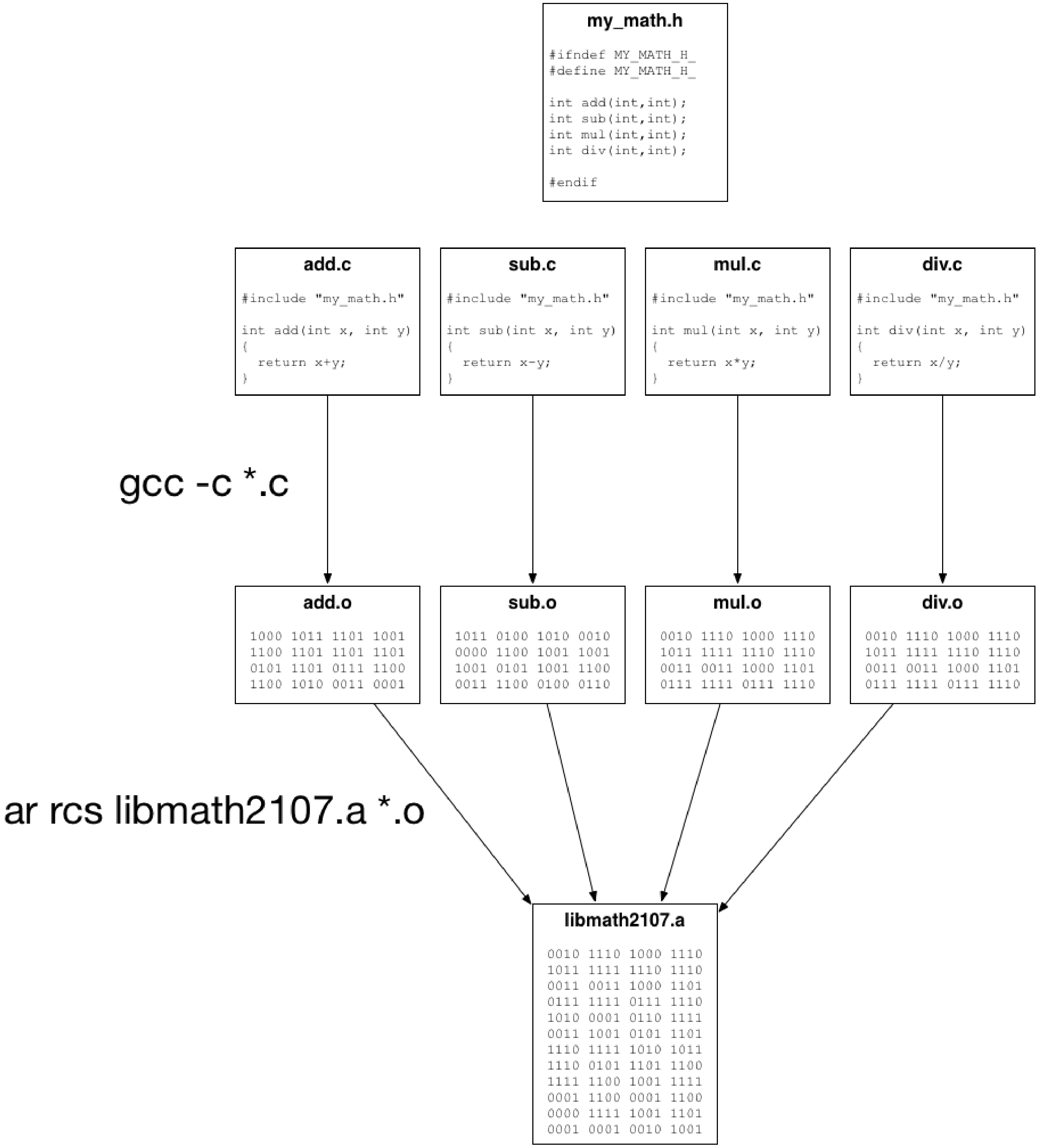
mul.o

```
0010 1110 1000 1110
1011 1111 1110 1110
0011 0011 1000 1101
0111 1111 0111 1110
```

div.o

```
0010 1110 1000 1110
1011 1111 1110 1110
0011 0011 1000 1101
0111 1111 0111 1110
```

which created the `.o` object files, which we bound together to form a library using the `ar` command.



my_math.h

```
#ifndef MY_MATH_H_
#define MY_MATH_H_

int add(int,int);
int sub(int,int);
int mul(int,int);
int div(int,int);

#endif
```

add.c

```
#include "my_math.h"

int add(int x, int y)
{
  return x+y;
}
```

sub.c

```
#include "my_math.h"

int sub(int x, int y)
{
  return x-y;
}
```

mul.c

```
#include "my_math.h"

int mul(int x, int y)
{
  return x*y;
}
```

div.c

```
#include "my_math.h"

int div(int x, int y)
{
  return x/y;
}
```

```
gcc -c *.c
```

add.o

```
1000 1011 1101 1001
1100 1101 1101 1101
0101 1101 0111 1100
1100 1010 0011 0001
```

sub.o

```
1011 0100 1010 0010
0000 1100 1001 1001
1001 0101 1001 1100
0011 1100 0100 0110
```

mul.o

```
0010 1110 1000 1110
1011 1111 1110 1110
0011 0011 1000 1101
0111 1111 0111 1110
```

div.o

```
0010 1110 1000 1110
1011 1111 1110 1110
0011 0011 1000 1101
0111 1111 0111 1110
```

```
ar rcs libmath2107.a *.o
```

libmath2107.a

```
0010 1110 1000 1110
1011 1111 1110 1110
0011 0011 1000 1101
0111 1111 0111 1110
1010 0001 0110 1111
0011 1001 0101 1101
1110 1111 1010 1011
1110 0101 1101 1100
1111 1100 1001 1111
0001 1100 0001 1100
0000 1111 1001 1101
0001 0001 0010 1001
```

Generating the Library

To create the library file, we use the `ar` command. The syntax is:

```
ar rcs NameOfTheLibraryFileToCreate listOfFilesToIncludeInTheLibrary
```

Don't forget that you're including the binary files, i.e., the `.o` files, not your `c` source files.

So suppose you have all of your function `.o` files in a single directory, and you'd like to call your library file `libstr2107.a`, you'd type:

```
ar rcs libstr2107.a *.o
```

To double-check, you can try:

```
ar t libstr2107.a
```

if you see a list of your `.o` files, you've done it right. The library file name should begin `lib` and should have a `.a` extension.

Using the library with your driver (10 points)

Write a very simple, basic program to test your functions. Suppose that it's in a file called `strtester.c`. To compile it, using your new string library, `libstr2107.a`, type:

```
gcc -o strtester strtester.c -LdirectoryWhereYouPutTheLibrary -lstr2107
```

so, if `libstr2107.a` is in your current directory, you'd type:

```
gcc -o strtester strtester.c -L. -lstr2107
```

Note that at the end of the line, it's `-l` (lower case `l`, not the number `1`), and it's just `str2107` not `libstr2107.a`.

The meaning of the command line is:

- **`-o strtester`** name the output file `strtester`
- **`-L.`** look for library files in the current directory
- **`-lstr2107`** link the contents of the file `libstr2107.a`

Deliverables

Please send your lab instructor either a single tar or zip file containing all of your `.c` files, your `.h` file and your string library file (your `.a` file). In order to help the TA keep track of everyone's files, please include your name in the name of the tar file.

Remember that you can find some notes on tarfiles by reading the other file attached named (`CIS2107_tar_and_zip`).

Some Tips, Reminders, etc.

String Literals

Here's another thing that you didn't need to worry about in Java that might cause some headaches in C. Remember that these two declarations are not exactly the same:

```
char *str01 = "What time does class end?";
char str02[] = "What time does class end?";
```

They both look the same when you print them, but if you try to modify `str01`, you'll get in trouble. The memory that `str01` points to is read-only. `str02` isn't, but it's only as much space as is required to hold the letters `'W'`, `'h'`, `'a'`, ..., and the null character (`'\0'`).

None of this is likely to be a big deal in any of the functions you write, but it might come up when you're trying to test a function. For example, something like:

```
char *str01 = "     Where's the remote control?     ";

...

strip(str01);
```

will give you a segmentation fault, but this doesn't necessarily mean that there's a problem in the `strip` function.

Videos:

Video: This is a short video about how to create static library and use it in Linux.