CISC 2000—Computer Science II
Fall, 2013

Programming Project #4: Strings, the Hard Way
Date Due: Thursday 14 November 2013

As C++-programmers, you have access to the Standard Template Library's `string` data type. C programmers aren't so fortunate; they have to work with a string representation consisting of a null-terminated array of `chars`. This means C uses a `0` byte to mark the end of the string. For example, the string `"hello"` (which has length 5) would be represented by the 6-element `char` array

'H'	'e'	'l'	'l'	'o'	0

The zero at the end might look a bit odd to you, since everything else has single quote marks. But a `char` is a small integer,[1] and so this really makes sense.

The way that you declare a string variable in C is

```
char greet[20];
```

If you want to initialize, this would look like

```
char greet[20] = "hello";
```

With this introduction, we can move on to the current project, which is based on some of the exercises in Chapter 18 of the text. The main idea is to implement several functions that manipulate C strings. You are *not* allowed to use any standard library functions.

1. Write a function

   ```
   void toupper(char* s)
   ```

   that replaces all lower-case characters in the C-style string `s` with their upper-case equivalents. For example,

   `"Hello, World!"` becomes `"HELLO, WORLD!"`

   This can made easier if you write two "helper functions":

[1] You can find the numerical values associated with characters by looking at the `ascii` manpage, i.e., by running the command `man ascii` in a shell window. To be really pedantic, the internal representation of `"hello"` (in decimal) would be

72	101	108	108	111	0

The hexadecimal representation is given by

0x68	0x65	0x6c	0x6c	0x6f	0x00

From this, we easily see that the binary representation is

01101000	01100101	01101100	01101100	01101111	00000000

- `bool islower(char c)`, which returns `true` if `c` is a lower-case letter, and `false` otherwise. Note that the lower-case letters are in the range `'a'-'z'`.
- `char toupper(char c)`, which returns the upper-case version of `c` if `c` is a lower-case letter, and `c` otherwise. (This can use `islower` described above.) Note that the upper-case letters are in the range `'A'-'Z'`, and (e.g.)

$$'\texttt{F}' = '\texttt{f}' - '\texttt{a}' + '\texttt{A}'$$

2. Write a function

```
char* strdup(const char* s)
```

that duplicates a C-style string. More precisely, it copies a C-style string `s` into memory that `strdup` allocates on the free store.

3. Write a function

```
char* cat_dot(const char *s1, const char* s2, const char* sep)
```

that returns a C-string consisting of `s1`, followed by `sep`, followed by `s2`. The default value of `sep` should be `"."`. Thus

`cat_dot("hello", "world")` should return the C-string `"hello.world"`

and

`cat_dot("hello", "world", "!?")` should return the C-string `"hello!?world"`.

This particular function should show you how painful it is to use C-strings, as opposed to C++-strings. Had you been asked to write `cat_dot` to work with C++-strings, it would simply be

```
// concatenate two C++-strings, with a separator
// default separator is a dot
string cat_dot(const string& s1, const string& s2,
               const string& sep=".")
{
   return s1 + sep + s2;
}
```

4. Section 18.6 of the text deals with *palindromes*, words (or phrases) that are spelled the same from both ends (e.g., *madam*, *eve*, *able was I ere I saw elba*,[2] and so forth. It's worth noting that most people ignore spaces, punctuation, and capitalization when determining whether a phrase is a palindrome; hence, *Madam, I'm Adam*[3] and *A man, a plan a canal—Panama!* are considered to be palindromes. We won't do this, since it spoils the pristine beauty of the concept.

[2] Napoleon's lament. Of course, "Elba" should be capitalized.
[3] Per the Bible, the first introduction ever made.

The text has two implementations of a function `is_palindrome()` that determines whether a string is a palindrome, returning a boolean value (`true` if its argument is a palindrome, `false` if not). The first implementation (for C++ strings) is on pp. 637–638, while the second (for C strings) is on pg. 640). Your task is to write a new version of `is_palidrome` for C strings, which works by making a backward copy of the string and then comparing; for example, it would take `"home"`, generate `"emoh"`, and compare those two strings to see that they are different, and so `"home"` isn't a palindrome. The C++ version of this would be the following:

```
// reverse a C++-string
string reverse(const string& s)
{
   string result;
   for (int i = s.length()-1; i >= 0; i--)
      result.push_back(s[i]);
   return result;
}

// is a C++-string a palindrome?
bool is_palindrome(const string& s)
{
   return s == reverse(s);
}
```

The C-version of `reverse` won't be too bad. However, you can't use == testing between C-strings; it will simply test for pointer equality. You'll either need to write a function that will test for equality of C-strings, or you'll need to embed this testing into your code for `is_palindrome`. I took the former approach. Alternatively, you can implement the function `strcmp()` described in Exercise 3 of Chapter 18; however, I don't want you to use the builtin version of `strcmp()`, which would defeat the whole purpose of this exercise.

Here are some pointers:[4]

1. Do all your work in the directory `~/private/cs2/proj4` (this should come as no surprise).
2. Your job will be to write a program `proj4.cc` that implements the four functions mentioned above, along with all necessary "helper functions" that I haven't already implemented.
3. The directory `~agw/class/cs2/share/proj4` has the following goodies:
   - A `Makefile`, which you should copy to your working directory. It works in the usual way; see previous project handouts (or the brief documentation at the top of the `Makefile`) for details.
   - A "stub version" of `proj4.cc`, which you should copy to your working directory. It contains function prototypes, the *complete* definition of `main()`, and the full definitions of two helper functions:
     - `strlen()` is used to find the length of a C-string. My implementation of `cat_dot()` and `reverse()` both used this function. Yours will probably do likewise.

[4] I can never resist a bad pun; sorry.

  - `get_c_string()` gets a line of input and stuffs it into a C-string. I use it in `main()`. You don't need to worry about it too much.

  It also contains stub versions of the functions you need to implement, as well as of various helper functions that I found to be useful. Your job mainly consists of completing the implementations of same; don't forget to replace the stub version of the initial comment that describes each function!

- A working version of `proj4`, called `proj4-agw`. Run it in the usual way; see previous project handouts for details.

Deliverables: Please turn in a *clean* typescript consisting of

- a listing of `proj4.cc`,
- evidence of correct compilation (e.g., `make clean` followed by `make`),
- a sample run of `proj4`.

Send it to me via email, in the usual way; see previous project handouts for details.

Good luck!