Problem Set 1

Due April 23, 2009, 11:59PM

In this problem set, you'll get a chance to practice with C strings and the preprocessor. In any question dealing with C strings, you can assume that the string is allocated somewhere in the heap (i.e. that you can modify its contents without causing a segmentation fault). As always, feel free to email me or to drop by after lecture if you have any comments or questions.

Problem 0

When working with C++ `string`s, the `erase` function can be used as `myString.erase(n)` to erase all characters in the string starting at position `n`, where `n` is assumed to be within the bounds of the string. Write a function `TruncateString` that accepts a `char *` C-style string and an index in the string, then modifies the string by removing all characters in the string starting at that position. You can assume that the index is in bounds. *(Hint: Do you actually need to remove the characters at that position, or can you trick C++ into thinking that they're not there?)*

Problem 1

Many C++ compiler vendors provide nonstandard C string manipulation functions along with the standard C++ functions.* One such function is `strdup`, which accepts as input a C-style string and returns a dynamically-allocated copy of that string. As a warm-up to some of the later problems, without using `strdup`, write a function `StringDuplicate` which accepts a single C string as input, then returns a dynamically-allocated copy of that string. You should use `new[]` to allocate memory for the string.

Problem 2

There are several ways to iterate over the contents of a C string. For example, we can iterate over the string using bracket syntax using the following construct:

```
int length = strlen(myStr); // Compute once and store for efficiency.
for(int k = 0; k < length; k++)
    myStr[k] = /* ... */
```

Another means for iterating over the C string uses pointer arithmetic and explicitly checks for the terminating null character. This is shown below:

```
for(char* currLoc = myStr; *currLoc != '\0'; ++currLoc)
    *currLoc = /* ... */
```

Write a function `CountFrequency` which accepts as input a C-style string and a character, then returns the number of times that the character appears in the string. You should write this function two ways – once using the first style of `for` loop, and once using the second.

---

* My personal favorite is gcc's `strfry`, which randomly scrambles the characters in a string. Thanks to Roy Frostig for pointing this one out.

Problem 3

Write a function `GetSubstring` which accepts three parameters – a C-style string, a start position, and a length – and returns a dynamically-allocated C string containing the subsequence of the source string containing characters in the range [start, start + length). You may assume that the start and end positions are contained within the bounds of the string. However, you should not use any standard library functions to implement this function.

Problem 4

Another common nonstandard addition to the C string library is a function `strcasecmp`, which returns how two strings compare to one another case-insensitively. For example,
`strcasecmp("HeLlO!", "hello!")` would return zero, while
`strcasecmp("Hello", "Goodbye")` would return a negative value because `Goodbye` alphabetically precedes `Hello`.

`strcasecmp` is not available with all C++ compilers, but it's still a useful function to have at your disposal. While implementing a completely-correct version of `strcasecmp` is a bit tricky (mainly when deciding how to compare letters and punctuation symbols), it is not particularly difficult to write a similar function called `StrCaseEqual` which returns if two strings, compared case-insensitively, are equal to one another.

Without using `strcasecmp`, implement the `StrCaseEqual` function. It should accept as input two C-style strings and return whether they are exactly identical when compared case-insensitively. The `toupper`, `tolower`, or `isalpha` functions from `<cctype>` might be useful here; consult a C++ reference for more details. To give you more practice directly manipulating C strings, your solution should not use any of the standard library functions other than the ones exported by `<cctype>`.

Problem 5

The `memset` function exported by `<cstring>` accepts as input three arguments – a pointer to a region in memory, a character value, and a number of copies – then writes that many copies of the specified character to that region in memory. Note that unlike the string manipulation routines we covered in class, `memset` does *not* write a terminating null to the end of the range.

Using `memset`, write a function `CreateRepetitiveString` which accepts as input a character and a length, then returns a dynamically-allocated C string consisting of that many copies of the specified character.

Problem 6

Modify the `TruncateString` function you wrote for Problem 0 such that the function raises an `assert` error if the truncation index is out of bounds. That is, if you try to truncate a 5-character string starting at character 17, `TruncateString` should report the error. You should, however, allow the user to truncate a string of length *n* at position *n*, which is effectively a no-op.

Problem 7

Suppose that you are designing a control system for an autonomous vehicle in the spirit of the DARPA Grand Challenge. As part of its initialization process, the program needs to call a function named `InitCriticalInfrastructure()` to set up the car's sensor arrays. In order for the rest of the program to respond in the event that the startup fails, `InitCriticalInfrastructure()` returns a `bool` indicating whether or not it succeeded. To ensure that the function call succeeds, you check the return value of `InitCriticalInfrastructure()` as follows:

```
assert(InitCriticalInfrastructure());
```

During testing, your software behaves admirably and you manage to secure funding, fame, and prestige. You then compile your program in release mode, install it in a production car, and to your horror find that the car immediately drives off a cliff. Later analysis determines that the cause of the problem was that `InitCriticalInfrastructure` had not been called and that consequently the sensor array had failed to initialize.

Why did the release version of the program not call `InitCriticalInfrastructure`? How would you rewrite the code that checks for an error so that this problem doesn't occur?

Problem 8

How long did this problem set take you? How hard was it? Did the questions help you get a better understanding of the material? Any suggestions for future problem sets?

Deliverables

To submit the assignment, email any files you've created to htiek@cs.stanford.edu. If possible, please send all answers either in a plain-text format (C++ source files are plain text) or as a PDF.

Good luck!