KAIST
EE209: Programming Structures for EE

Assignment 2: String Manipulation

(Acknowledgment: This assignment is borrowed and modified from Princeton's COS 217)

Purpose

The purpose of this assignment is to help you learn/review (1) arrays and pointers in the C programming language, (2) how to create and use stateless modules in C, (3) the "design by contract" style of programming, and (4) how to use the GNU/UNIX programming tools, especially `bash`, `emacs` (or any other editor of your choice), `gcc`, and `gdb`.

Rules

Implement the string functions listed by the Table in Part 1. and Part 2 is implementing 'Simple Grep', which is worth 50% of this assignment.

You will get an **extra** 10% of the full score if you implement the Part 1 only with pointer notation when you access the character. See the Extra credit section below (Extra Credit for Part 1).

Background

As you know, the C programming environment provides a standard library. The facilities provided in the standard library are declared in header files. One of those header files is string.h; it contains the declarations of "string functions," that is, functions that perform operations on character strings. Appendix D of the King textbook, Appendix B3 of the Kernighan and Ritchie textbook, and the UNIX "man" pages describe the string functions. The string functions are used heavily in programming systems; certainly any editor, compiler, assembler, or operating system created with the C programming language would use them.

Overview of Your Task

Your task in this assignment is to use C to create the "Str" module that provides string manipulation functions. Specifically, design your Str module so that each function behaves the same as described below. Your task in this assignment is twofold.

[Part 1] Read the description of the basic string library functions carefully, and implement each function. The basic functions are most commonly used standard string functions. Each function should behave the same as its corresponding standard C function.

[Part 2] Implement a simplified version of `grep` using `Str` functions. Read the provided file that contains skeleton code carefully, edit the file to make it process the required functionality.

NOTE: We provide skeleton code and several utilities (e.g., Makefile and tests) in assignment2.tar.gz. Please download it to your directory before starting your project and check `README.md`.

Part 1: The Basic `Str` Function Implementation

Your task for the first part is to use C to implement seven basic string manipulation functions: `StrGetLength(), StrCopy(), StrCompare(), StrFindChr(), StrFindStr(), StrConcat(), and StrToLong()`. Those seven functions should follow the format of the corresponding standard C library functions. You can easily find the function's description and operation in the UNIX "man" page.

The following table shows the required basic string functions in Part 1 and their corresponding function names in the standard C library.

`Str` Function	Standard C Function	Man page link
`StrGetLength`	`strlen`	strlen man page
`StrCopy`	`strcpy`	strcpy man page
`StrCompare`	`strcmp`	strcmp man page
`StrFindChr`	`strchr`	strchr man page
`StrFindStr`	`strstr`	strstr man page
`StrConcat`	`strcat`	strcat man page
`StrToLong`	`strtol`	strtol man page

Use the `Str` module's *interface* in a file named `str.h`, and place your `Str` function definitions in `str.c`.

Note that your Str functions **should not** call any of the standard string functions. In the context of this assignment, pretend that the standard string functions do not exist. However, your functions may call each other, and you may define additional (non-interface) functions.

Design each function definition so it calls the `assert` macro to validate the function's parameters. In that way, your `Str` functions should differ from the standard C string functions. Specifically, design each function definition to assert that each parameter is not `NULL`. See the note below for more information of the `assert()` macro function.

Beware of type mismatches. In particular, beware of the difference between type size_t and type int: a variable of type size_t can store larger integers than a variable of type int can. Your functions should (in principle) be able to handle strings whose lengths exceed the capacity of type int. Also beware of type mismatches related to the use of the const keyword.

Assume the third parameter in `StrToLong()` is always 10 (base == 10). Otherwise, the function should simply returns 0. Also, the function does not have to set `errno` in case of an error, but the return value must be identical as that of `strtol()` even in the case of an error (e.g., 0, `LONG_MIN`, `LONG_MAX`). Use `#include <limits.h>` to use `LONG_MIN` and `LONG_MAX` in your code. BTW, you should not use a similar function like `atoi(), atol(), ...` to implement this function.

Some hints on implementation of `StrToLong()`

1. How to skip spaces?

Use `isspace()` to check if the current character is one of spaces. For example, `while (*p && isspace(*p)) p++;` would move p to point to a non-space character in the string, assuming p initially points to the input string. For each character, make sure to check if *p is not '\0' (or simply 0). If p points to a null charater, you have reached the end of the string.

2. How to convert a number string to an int value?

Refer to the following code. p points to a number string, and x is the current result (with initial value of 0). For each digit, you need to multiply x with 10 and add the digit to x.

```
int x = 0;
char *p = "3456";
int temp;

while (*p && isdigit(*p)) {
   temp = *p - '0';
    x = 10 * x + temp;
   p++;
}
printf("number =%d\n", x); // it will print out 3456
```

Of course, you should use `long int` as the result type for `StrToLong()`, and take care of over/underflows while handling the `'+'` and `'-'` sign as well.

Extra Credit for Part 1: Implement functions with pointer notation

There are various ways to implement the functions in Part 1. Especially, you can access the character by pointer dereferencing like `*pcSrc` or by using an array notation such as `pcSrc[uiLength]` .

Here are two examples of `StrGetLength()` implementation. The first code implements the StrGetLength() function with the array notation; it traverses each given string or accesses the character using an index relative to the beginning of the string. However, with the pointer notation, the second version traverses each given string using an incremented pointer.

```
size_t StrGetLength(const char pcSrc[]) /* Use array notation */
{
   size_t uiLength = 0U;
   assert(pcSrc != NULL);
   while (pcSrc[uiLength] != '\0')
      uiLength++;
   return uiLength;
}
```

```
size_t StrGetLength(const char *pcSrc) /* Use pointer notation */
{
   const char *pcEnd;
   assert(pcSrc != NULL);
   pcEnd = pcSrc;
   while (*pcEnd != '\0') /* note that *(pcSrc + uiLength) is valid but is "NOT" acceptable as pointer notation */
      pcEnd++;
   return (size_t)(pcEnd - pcSrc);
}
```

You can freely implement Part 1. However, **if you implement part 1 only with pointer notation, you can get extra 10% of the full score.** Please write your choice of implementation in the `readme` file. That is, specify if you used only the pointer notation for Part 1 in the `readme` file.

Test your Str Functions

We provide a test client (`client.c`) that compares the results of the `Str` and C standard library functions with various input. Please use this code for testing and debugging. You can compile the test client with `gcc209` as follows.

```
gcc209 -o client client.c str.c
```

Then, test each of your `Str` functions separately by providing a function name as an argument. For example,

```
./client StrCopy
```

will test `StrCopy`. Actually, the client accepts any one of the `Str` function names as a command-line argument:

```
./client [StrGetLength|StrCopy|StrCompare|StrFindChr|StrFindStr|StrConcat|StrToLong]
```

Note that passing all tests provided by the client does **not** mean that your function always behaves correctly. Please devise your own testing (e.g., by changing the client code) for more confidence. Note that we may use a different test client for grading.

Part 2: Simple Grep

`grep` is a popular UNIX tool that manipulates input strings. In this part, you implement a simplified version of `grep` called `sgrep`. `sgrep` searches for a given pattern in each line from stdin, and prints out the line to stdout if it matches the pattern.

```
./sgrep [pattern]
```

pattern can be a simple string (e.g., 'hello', 'kaist') or it can inlcude one or more '*' in the string. '*' matches zero or more repetition of any character (exceptine character ('\n')). For example, 'ka*is*t' can be matched with 'kaist', 'kaxxist', or 'kaaaisss ?kkt'.

Here are some usage examples of `sgrep` (you can also see the test files (`google_wiki.txt` and `microsoft.txt`) which are used in the following example):

```
$ ./sgrep "*og*e" < google.txt
Google Inc. is an American multinational technology company specializing
advertising technologies, search, cloud computing, and software.[5]
Google was founded by Larry Page and Sergey Brin while they were Ph.D.
students at Stanford University. Together, they own about 14 percent of
 supervoting stock. They incorporated Google as a privately held company on
 be evil".[9][10] In 2004, Google moved to its new headquarters in Mountain
View, California, nicknamed the Googleplex.[11] In August 2015, Google
Alphabet Inc. When this restructuring took place on October 2, 2015, Google
became Alphabet's leading subsidiary, as well as the parent for Google's

$ ./sgrep "s*t" < microsoft.txt
Microsoft Inc. is an American multinational technology company specializing
in Internet-related services and products. These include online
advertising technologies, search, cloud computing, and software.[5]
Most of its profits are derived from AdWords,[6][7] an online advertising
service that places advertising near the list of search results.
Microsoft was founded by Larry Page and Sergey Brin while they were Ph.D.
students at Stanford University. Together, they own about 14 percent of
its shares and control 56 percent of the stockholder voting power through
 supervoting stock. They incorporated Microsoft as a privately held company on
An initial public offering followed on August 19, 2004.
Its mission statement from the outset was "to organize the world's
and its unofficial slogan was "Don't
 be evil".[9][10] In 2004, Microsoft moved to its new headquarters in Mountain
View, California, nicknamed the Microsoftplex.[11] In August 2015, Microsoft
announced plans to reorganize its interests as a holding company called
Alphabet Inc. When this restructuring took place on October 2, 2015, Microsoft
became Alphabet's leading subsidiary, as well as the parent for Microsoft's
Internet interests.[12][13][14][15][16]

$ ./sgrep "s t" < microsoft.txt
announced plans to reorganize its interests as a holding company called
became Alphabet's leading subsidiary, as well as the parent for Microsoft's
```

Rules:

- Use your Str functions to implement the functionality. That is, you need to finish Part 1 first to get any credit for this task. You should not use any string function in string.h
- Assume each line (including a new line character ('\n') is no more than 1023 bytes. You should stop your program with a proper error message(`"Error: input line is too long\n"`) if you encounter a line larger than 1023 bytes.
- <u>A `pattern` argument refers to a sequence of any characters that may contain spaces or '*'. In case it contains spaces or '*', it must be enclosed with double quote (") characters (e.g., "hello world", "s*t"). The pattern can be empty (""), and an empty pattern matches any line. No need to handle an escape character in the pattern string for this assignment.</u>
- Assume a pattern string no more than 1023 bytes. You should stop your program with a proper error message(`"Error: pattern is too long\n"`) if you encounter a command-line argument that is too long.
- Any error message should be printed out to standard error (`stderr`). Use `fprintf(stderr, ...)` for that.
- Unlike Part 1, you don't have to add assert function while implementing `sgrep`. In other words, after printing out the error messages in error case, `sgrep` should be finished with EXIT_FAILURE as the return value. Details are written in skeleton code, so please read the comments in skeletion code (`sgrep.c`).

Tips:

- We provide a skeleton code file (`sgrep.c`). You can start with the file.

- `fgets` should be useful for reading a line from a file. 'man fgets' should give you more information.
- You are free to choose whatever implementation strategy, but we suggest using a recursive function for implementing '*'. In fact, it will require less then 20 lines of code to implement '*' with a recursive function. Well, a higher performance implementation should use another technique, but that's not necessary for this assignment. We won't penalize for low performance unless it's really slow (e.g., seconds of delay for a short text).

Logistics

Develop on lab machines using `emacs` (or any other editor of your choice) to create source code and gdb to debug.

For part 1, you implement the code `str.c`, which should contain the definitions of required functions.

For part 2, a skeleton C file is available here: sgrep.c. It implements the basic I/O and argument parsing, so all you need is fill out the rest of the functionalities with `Str` functions. Feel free to use this file as your starting point (of course, you don't have to use it), and change any part in the code if you'd like.

Create a `readme` text file that contains:

- Your name and your student ID
- A description of whatever help (if any) you received from others while doing the assignment, and the names of any individuals with whom you collaborated, as prescribed by the course "Policy" web page.
- *Is it possible for `StrCopy` to call `assert` to verify that the destination memory area specified by the caller is large enough? Explain.*
- (Optionally) Please write the your implementation method in `readme` text file whether you use pointer notation for the extra credit.
- (Optionally) An indication of how much time you spent doing the assignment.
- (Optionally) Your assessment of the assignment.
- (Optionally) Any information that will help us to grade your work in the most favorable light. In particular you should describe all known bugs.

Submission

Use Homepage Submission Link to submit your assignments. Your submission should be one gzipped tar file whose name is **YourStudentID_assign2.tar.gz** . For example, if your student ID is 20161234, please name the file as 20161234_assign2.tar.gz.

Your submission file need to include the following files:

- (Task 1) updated `str.c` file
- (Task 2) updated `sgrep.c` file
- `readme` text file
- Observance of Ethics. Sign on the document, save it into a PDF file, and submit it.

Your submission file should look like this:

- 20211234_assign2.tar.gz
  - str.c
  - sgrep.c
  - readme
  - EthicsOath.pdf

Grading

If your submission file does not contain the expected files, or your code cannot be compiled at `eelab5` with `gcc209`, we cannot give you any points. Please double check before you submit.

We will grade your work on quality from the user's point of view and from the programmer's point of view. To encourage good coding practices, we will deduct points if `gcc209` generates warning messages.

From the user's point of view, your module has quality if it behaves as it should.

In part, style is defined by the rules given in *The Practice of Programming* (Kernighan and Pike), as summarized by the Rules of Programming Style document. These additional rules apply:

Names: You should use a clear and consistent style for variable and function names. One example of such a style is to prefix each variable name with characters that indicate its type. For example, the prefix `c` might indicate that the variable is of type `char`, `i` might indicate `int`, `pc` might mean `char*`, `ui` might mean `unsigned int`, etc. But it is fine to use another style -- a style which does not include the type of a variable in its name -- as long as the result is a readable program.

Line lengths: Limit line lengths in your source code to 72 characters. Doing so allows us to print your work in two columns, thus saving paper.

Comments: Each source code file should begin with a comment that includes your name, the number of the assignment, and the name of the file.

Comments: Each function should begin with a comment that describes what the function does from the caller's point of view. The function comment should:

- Explicitly refer to the function's parameters (by name) and the function's return value.
- State what, if anything, the function reads from standard input or any other stream, and what, if anything, the function writes to standard output, standard error, or any other stream.
- State which global variables the function uses or affects.
- Appear in both the interface (.h) file for the sake of the *clients* of the function and the implementation (.c) file for the sake of the *maintainers* of the function.

Parameter Validation: Validate function parameters via asserts whenever possible.

Please note that you might not get a full credit even if you pass the test with your `testclient`. TAs might use another `testclient` to test functionality and robustness of your implementation.

Note: assert and NDEBUG

assert is a macro implementations of assertion, used for verifying the conditions. If the condition is true, it does nothing. However, if the conditions is FALSE, it displays an error messages and aborts the running program.

In this assignment, you should validate the function's parameters with `assert`. When you try to check whether your `Str` functions validate the given parameters correctly or not, the aborted program may annoy you. In that case, you can add `NDEBUG` macro in your source file to ignore the assert functions. Otherwise, you can also add -D NDEBUG argument to the `gcc`. The `-D NDEBUG` argument commands `gcc209` to define the `NDEBUG` macro, just as if the preprocessor directive `#define NDEBUG` appeared in the specified .c file(s). Following OBcommands are example of disabling assert using -D option

```
gcc209 -D NDEBUG sgrep.c str.c -o sgrep
```

If NDEBUG is defined as a macro name in the source file, the assert macro is defined as `((void)0)`, which means that the assert macro will be ignore.

Note: Using Idioms

C programmers sometimes use idioms that rely on the fact that the null character ('\0'), the `NULL` address, the integer 0, and the logical concept FALSE have the same representation. You may use those idioms. For example, you may define `StrGetLength` like this:

```
size_t StrGetLength(const char pcSrc[])
{
   size_t uiLength = 0U;
   assert(pcSrc); /* NULL address, 0, and FALSE are identical. */
   while (pcSrc[uiLength]) /* null character and FALSE are identical. */
      uiLength++;
   return uiLength;
}
```

or like this:

```
size_t StrGetLength(const char *pcSrc)
{
   const char *pcEnd;
   assert(pcSrc); /* NULL address, 0, and FALSE are identical. */
   pcEnd = pcSrc;
   while (*pcEnd) /* null character and FALSE are identical. */
      pcEnd++;
   return (size_t)(pcEnd - pcSrc);
}
```

But you are not required to use those idioms. In fact, we recommend that you avoid the use of idioms that adversely affect understandability.

Note: The `const` Keyword and `StrSearch`

The use of the `const` keyword within `StrFndStr` is tricky, as this question-and-answer sequence indicates.

Question

According to the man pages, the parameters of `strstr` are of type `const char*`. That implies that the parameters of `StrFndStr` also should be of type `const char*`. Why are they not of type `char*`?

Answer

Suppose you were to define `StrFindStr` like this:

```
char *StrFindStr(char *pcHaystack, char *pcNeedle)
{
   ...
}
```

Further suppose the client then calls `StrFindStr` like this:

```
const char *pcBig = "hello";
const char *pcSmall = "lo";
 ...
 ... StrFindStr(pcBig, pcSmall) ...
 ...
```

(Note that's a perfectly reasonable way to call the function.) In that case the compiler, noting that `pcBig` is of type `const char*` and that `pcHaystack` is of type `char*`, would generate a warning on the function call. Thus `pcHaystack` (and `pcNeedle`) should be of type `const char*`.

Question

According to the man pages, the return type of `strstr` is `char*`. That implies that the return type of `StrFindStr` also should be of type `char*`. Why is the return type not `const char*`?

Answer

Suppose you were to define `StrFindStr` like this:

```
const char *StrFindStr(const char *pcHaystack, const char *pcNeedle)
{
   ...
}
```

Further suppose the client then calls `StrFindStr` like this:

```
char *pcBig = "hello";
char *pcSmall = "lo";
char *pc;
 ...
pc = StrFindStr(pcBig, pcSmall);
 ...
```

(Note that's a perfectly reasonable way to call the function.) In that case the compiler, noting that `pc` is of type `char*` and that the value returned by `StrFindStr` is of type `const char*`, would generate a warning on the assignment statement. Thus the return type of `StrFindStr` should be `char*` and should not be `const char*`.

Question

Within the definition of `StrFindStr`, I decided to define a local variable named `pc` that "points into" the first of the two given strings, as indicated by this code:

```
char *StrFindStr(const char *pcHaystack, const char *pcNeedle)
{
   ...
   pc = pcHaystack;
   ...
   /* Increment pc so it points to the appropriate character. */
   ...
   return pc;
}
```

If I define `pc` to be of type `char*`, then the assignment statement generates a warning. If I define `pc` to be of type `const char*`, then the `return` statement generates a warning. How can I resolve that problem?

Answer

Unfortunately, C provides no elegant solution. We recommend that you define `pc` to be of type `const char*` so the assignment statement is warningless. Then use a cast operator in the `return` statement:

```
return (char*)pc;
```

to explicitly inform the compiler to not generate a warning. C programmers refer to that solution as "casting away the constness" of the variable. Sadly, often that inelegant technique is unavoidable.