Fastest way of removing a substring from a string

Taking a break from my C++ series, I recently reviewed the code in this question. I ended up writing my own separate derivation of the function so that I co
remove all of the substrings from a certain string.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

#define NUM_CYCLES 10000000

void rmSubstr(char *str, const char *toRemove)
{
    size_t length = strlen(toRemove);
    while((str = strstr(str, toRemove)))
    {
        memmove(str, str + length, 1 + strlen(str + length));
    }
}

clock_t runTest(void (*fn)(char*, const char*), char *str, char *substr)
{
    clock_t start = clock();
    for (int i = 0; i < NUM_CYCLES; ++i)
    {
        fn(str, substr);
    }
    return clock() - start;
}

int main(void)
{
    char* str1 = strdup("test this string out");
    char* str2 = strdup("test this ..........slightly........... longer thing which
contains a sub string out");
    char* str3 = strdup("string");
    char* str4 = strdup("this string is not a string test");

    printf("Function ran in %lu clock cycles\nProduced output: %s\n\n", runTest(rmSubstr,
str1, " string"), str1);
    printf("Function ran in %lu clock cycles\nProduced output: %s\n\n", runTest(rmSubstr,
str2, " string"), str2);
    printf("Function ran in %lu clock cycles\nProduced output: %s\n\n", runTest(rmSubstr,
str3, " string"), str3);
    printf("Function ran in %lu clock cycles\nProduced output: %s\n\n", runTest(rmSubstr,
str4, " string"), str4);
}
```

Generates on my MacBook Pro (Intel Core i5 2.4 GHz with 2 cores, 4 GB RAM, running Mac OS X 10.10 beta 4):

```
Function ran in 319220 clock cycles
Produced output: test this out

Function ran in 1293377 clock cycles
Produced output: test this ..........slightly........... longer thing which contains a
sub out

Function ran in 198460 clock cycles
Produced output: string

Function ran in 448832 clock cycles
Produced output: this is not a test

Program ended with exit code: 0
```

The only thing I would like to be reviewed in this question is the speed of the `rmSubstr` function. Other comments are welcome, but alone will not be accep
Also, please include the output of the program with your answer so that I can get a feeling for how much my function was improved.

performance c strings search rags-to-riches

edited Apr 13 '17 at 12:40
Community ♦
1

asked Oct 17 '14 at 21:38
syb0rg
16.8k 7 90 17

The current fastest solution will be marked as the accepted answer, and is subject to change if beaten. – syb0rg Oct 27 '14 at 0:24

5 Answers

In surveying these answers I surmised that the best solution would be to do partial memory copies rather than to-end-of-line copies. This would pay dividends in the event of multiple matches.

As a result, I put together this method:

```
void rolfl(char *str, const char *toRemove) {
    if (NULL == (str = strstr(str, toRemove)))
    {
        // no match.
        //printf("No match in %s\n", str);
        return;
    }

    // str points to toRemove in str now.
    const size_t remLen = strlen(toRemove);
    char *copyEnd;
    char *copyFrom = str + remLen;
    while (NULL != (copyEnd = strstr(copyFrom, toRemove)))
    {
        //printf("match at %3ld in %s\n", copyEnd - str, str);
        memmove(str, copyFrom, copyEnd - copyFrom);
        str += copyEnd - copyFrom;
        copyFrom = copyEnd + remLen;
    }
    memmove(str, copyFrom, 1 + strlen(copyFrom));
}
```

It finds the location of data to keep, and moves each 'keep' span just once. No data is ever moved more than once.

It works by keeping a cursor of the start of the copy-zone, the start of the following match (if any), and then it copies that region on to the end of previous content (advancing that copyTo variable as needed).

It was only after I implemented the solution myself that I realized how similar the routine was to Edwards. I do prefer my naming though.

The significant performance-affecting difference is that I only have to perform the `strlen(src)` on the final (shortest) span of unmatching code. That `strlen(src)` is essentially the only difference I can see in the effect of the algorithm.... and, this difference will become more and more apparent as the input String size increases.

When I run it through Edward's harness (I ran without William's code...), I get:

```
totaltime syb0rg = 1710000
totaltime 200_success = 1590000
totaltime rolfl = 1070000
totaltime janos = 1830000
totaltime Edward = 1100000
Winner is rolfl
```

A subsequent run gives:

```
totaltime syb0rg = 1750000
totaltime 200_success = 1610000
totaltime rolfl = 1050000
totaltime janos = 1840000
totaltime Edward = 1080000
Winner is rolfl
```

Now, that difference is small, I realize, but it is real ;-)

Also, admittedly, occasionally when I run it Edward wins....

(compiled with `gcc -o go -Wall -D_GNU_SOURCE -O3 -std=c99 main.c` on an AMD linux machine).

When I run it on an intel machine (core i7 4770):

```
totaltime syb0rg = 581688
totaltime 200_success = 592600
totaltime rolfl = 448699
totaltime janos = 787115
totaltime Edward = 472100
Winner is rolfl
```

answered Oct 27 '14 at 18:41
rolfl
89.2k 13 185 388

Nice! It also has the advantage over mine in that if a string ends with the target, mine fails to chop it, but this routine performs correctly. – Edward Oct 29 '14 at 0:39

It should be possible to optimize a global search-and-replace by doing all the moves in one pass.

```
void rmSubstr(char *str, const char *toRemove)
{
    size_t length = strlen(toRemove);
    char *found,
         *next = strstr(str, toRemove);

    for (size_t bytesRemoved = 0; (found = next); bytesRemoved += length)
    {
        char *rest = found + length;
        next = strstr(rest, toRemove);
        memmove(found - bytesRemoved,
                rest,
                next ? next - rest: strlen(rest) + 1);
    }
}
```

Output using your test code:

```
Function ran in 351851 clock cycles
Produced output: test this out

Function ran in 1349681 clock cycles
Produced output: test this ...........slightly............ longer thing which contains a
sub out

Function ran in 197570 clock cycles
Produced output: string

Function ran in 416682 clock cycles
Produced output: this is not a test

Program ended with exit code: 0
```

edited Oct 18 '14 at 17:32
Jamal ♦
29.6k 11 111 220

answered Oct 17 '14 at 23:34
200_success ♦
119k 14 135 388

In practice, there doesn't seem to be a significant performance difference. – 200_success ♦ Oct 17 '14 at 23:42

1 This is a much better solution than either the original or the other two solutions. With my test string (see my answer) it is twice as fast. It took me a while to see why, but (for the benefit of other readers) it is because your solution does only two `strlen` calls (as do mine and @janos') and its `memmove` calls move only the bytes between the detected sub-strings, not the who string following each detected sub-string. Very neat! – William Morris Oct 18 '14 at 20:24

It might be useful to modify the function to return the number of substrings removed. – 200_success ♦ Oct 18 '14 at 21:16

What fun! Since I'm late to this party, first, I decided to improve the text fixture. I created a generic data structure to compare all routines.

```
typedef struct delstring
{
    void (*routine)(char *, const char *);
    const char *name;
    unsigned long elapsed;
} delstring;
```

Then I created a static array of them.

```
static delstring functions[] = {
    { syb0rg,       "syb0rg", 0 },
    { WilliamMorris,"William Morris", 0 },
    { success_200,  "200_success", 0 },
    { janos,        "janos", 0 },
    { edward,       "Edward", 0 },
    { NULL, NULL, 0},
};
```

Then I put all of the test strings into yet another structure:

```
static const char *teststring[] = {
    "test this string out",
    "test this ...........slightly............ longer thing which contains a sub string
out",
    "string",
    "this string is not a string test",
    "foo string string string string stringbar",
    "f stringa stringb stringu stringl stringo stringu strings string!",
    NULL
};
```

Finally, the rewritten `main` :

```
int main(void)
{
    const char **orig;
    delstring *test;
    for (test = functions; test->name; ++test)
    {
        for (orig = teststring; *orig; ++orig)
        {
            char *str = strdup(*orig);
            clock_t time = runTest(test->routine, str, " string");
            test->elapsed += time;
            printf("\"%s\",%lu,\"%s\"\n", test->name, time, str);
            free(str);
        }
    }
    delstring *winner = functions;
    for (test = functions; test->name; ++test)
    {
        printf("totaltime %s = %lu\n", test->name, test->elapsed);
        if (test->elapsed < winner->elapsed)
            winner = test;
    }
    printf("Winner is %s\n", winner->name);
}
```

You can infer, no doubt, that the various routines within the `functions` array come from the other answers and from the original, plus my function which is here:

```
void edward(char *dst, const char *toRemove)
{
    char *chop, *src;
    if (NULL == (dst = strstr(dst, toRemove)))
        return;
    size_t length = strlen(toRemove);
    src = dst + length;
    char *end = src + strlen(src);
    while((chop = strstr(src, toRemove)))
    {
        memmove(dst, src, chop-src+1);
        dst += chop-src;
        src = chop+length;
    }
    if (src < end) {
        memmove(dst, src, end-src+1);
    }
}
```

The (abbreviated) results on my machine:

```
totaltime syb0rg = 1035976
totaltime William Morris = 1192584
totaltime 200_success = 969344
totaltime janos = 1221978
totaltime Edward = 824009
Winner is Edward
```

Now the actual code review part:

Remember to `free` memory

The `strdup` function makes a copy that must be released with `free` . The revised `main` shown above does that, but the original code did not.

Bail early to save time

There isn't any need to do any copying if the target string doesn't exist even once within the string. In the code I showed above, there's an early bailout in the event that the string isn't found.

Help the optimizer

The compiler's optimization can likely move loop invariants outside the loop, but you can help by doing things such as invoking `strlen` once and storing the result. Your code does this, but it's worth checking on your computer and your compiler.

Copy as little as possible

If there are many copies of the target string within the passed string, the original code will copy the entire tail (that is, everything after the match) many times. The better approach, in terms of speed, is to copy only the portion after then end of one match to the beginning of the next.

Consider consolidating moves

What I tried, but that did not actually result in faster times on my machine, was to also optimize so that multiple contiguous matches would skip the copy and simply increase the copy size. That is, if the target was "Mad " and the string was "It's a Mad Mad Mad Mad World" one could simply move "World" once rather than moving "Mad Mad Mad World" and then "Mad Mad World", etc. You might still want to try that on your machine with your compiler to see if it actually works there.

Make sure you're measuring something meaningful

The original test code (which I did not change) first passes in a copy of the original test string, but then each subsequent iteration within `runTest` performs the operation on the already modified string! This is largely why my routine (which is the only one that bails out early if no instance of the target was found) fares so well. *You're not measuring the true performance of the routines!*

answered Oct 24 '14 at 1:56
Edward
40.5k 3 68 182

First of all, `gcc` gives me a warning for every occurrence of the `" string"` used as the parameter of `runTest`:

```
warning: conversion from string literal to 'char *' is deprecated
      [-Wdeprecated-writable-strings]
```

It's quite annoying when I recompile and rerun, I recommend to do it this way instead:

```
char* toRemove = strdup(" string");
printf("Function ran in %lu clock cycles\nProduced output: %s\n\n", runTest(rmSubstr,
str1, toRemove), str1);
```

I was trying something similar as @William Morris:

```
void rmSubstr(char *str, const char *toRemove)
{
    size_t toRemoveLength = strlen(toRemove);
    size_t origLength = strlen(str);
    char* work = str;
    int offset, remainingLength;
    while ((work = strstr(work, toRemove)))
    {
        offset = work - str;
        remainingLength = 1 + origLength - offset - toRemoveLength;
        memcpy(work, work + toRemoveLength, remainingLength);
    }
}
```

The idea is to move the `strlen` out of the loop, and calculate the remaining length using arithmetics. I don't get how `strlen` does it faster, but apparently it does. Even for longer test strings like this:

```
// after replacement, gives "oh boy"
char* str5 = strdup("oh string string string string string string string string string
string string string string string string string string string string string string
string string string string string string string string string string string string
string string string string string string string string string string string boy");
```

Puzzler. Note that the technique will always be slower for cases when there is nothing to remove, as in that case the `strlen` up front will never be used.

edited Apr 13 '17 at 12:40
Community ♦
1

answered Oct 18 '14 at 14:56
janos ♦
91.9k 12 109 328

Testing your version, it is about the same speed as mine and somewhat faster than the original. The version by @200_success is twice as fast - see my comment above. – William Morris Oct 18 '14 at 20:25

You can speed it up by removing the `strlen` from the loop:

```
static void rmSubstr(char *s, const char *sub)
{
    const size_t sub_len = strlen(sub);
    size_t s_len = strlen(s) + 1;
    for (char *t; (t = strstr(s, sub)) != NULL; s = t) {
        s_len -= (sub_len + (size_t) (t - s));
        memmove(t, t + sub_len, s_len);
    }
}
```

I tested it using:

```
static clock_t runTest(void (*fn)(char*, const char*), const char *orig, const char
*substr)
{
    clock_t start = clock();
    char* str = strdup(orig);
    size_t len = strlen(orig);

    for (int i = 0; i < NUM_CYCLES; ++i)
    {
        memcpy(str, orig, len+1);
        fn(str, substr);
    }
    //printf("Result: %s\n", str);
    return clock() - start;
}
```

in which the test string is re-used on each loop unmodified.

And used a `main` of

```
int main(void)
{
    const char* str4 = "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test"
        "this string is not a string test";
    printf("New function ran in %lu clock cycles\n", runTest(rmSubstr, str4, " string"));
    printf("Orig function ran in %lu clock cycles\n", runTest(rmSubstr_orig, str4, "
string"));
}
```

Using 10x fewer loops than your original, this gives:

```
New function ran in 4278925 clock cycles
Orig function ran in 5581592 clock cycles
```

edited Oct 18 '14 at 17:36

answered Oct 18 '14 at 13:06

William Morris
8,644 12 41