Lecture 6: Chars and C-Strings (take 2)

Monday, January 22, 2024

Reading: Reader: Ch 4, C Primer, Ch 7, C Strings, K&R (1.9, 5.5, Appendix B3), or Essential C section 3 for C-strings and string.h library functions.

Lecturer: Chris Gregg
Today's Topics

• Logistics
  • Assign1 — Due Wednesday at 11:59pm
• Reading: Reader: *C Primer, C Strings*, K&R 1.9, 5,5, Appendix B3
  • Chars
    • ctype library
  • C-Strings
    • How strings are laid out in memory
    • The string.h library
First Feedback!

We received our first "High Frequency Feedback" and I wanted to take a couple of minutes to discuss some of the comments.

From "Is anything from class still confusing to you?"

"yes unix is confusing and the lack of a tutorial on how to use emacs/vim is kind of rude"

Yes, Unix commands do take a while to learn! You'll get there! We do have tutorials for both emacs and vim, by the way (see the Handouts tab on the website).

"assignments are very tedious"

Well — yes, assignments are challenging. I don't know that I would use the word "tedious", but this is a hard class, with long, hard assignments.
"I would love to see more in-class examples."

This is a tough one — I have to make a call to teach you content and show you examples. I do my best to accomplish both, but we are certainly time-limited in lecture. One place to see lots of examples is in lab, which is where you can learn a lot by digging in and working with code.

"Sometimes I feel like Chris jumps to quickly into things that we might not have prior experience with (I would've appreciated a slower introduction to the GDB debugger, Unix, and some of the basics of C)."

Yes, I wish we had time to take it a bit slower. Alas. Again, lab is (and A0 was) your friend here.
"Lectures move very quickly and we don't always finish the slides, so I'm not always sure what information I am accountable for on hw or before the next lecture."

Yes, sometimes I don't get to all the slides. I will always get to the rest of those slides in some form — most often, in the next lecture, but very infrequently I'll post a follow-up video. I don't want this class to creep into the "you need to watch 3 hours of lecture per class period!" so I want to be careful with extra videos.

"I am still confused on some of the bit masking material."

Lab and assignment 1 (where you learn a lot just doing it) are going to be very helpful here. And come to office hours!
Learning gdb as soon as you can is going to pay off immensely this quarter!

This will help your debugging, and will help in office hours when your CA says, "what does the debugger say? Where is your bug?"

Today, we will see lots of debugging in gdb, as well.
ChatGPT

- As I mentioned in Lecture 1, we are checking for the use of ChatGPT and the like.
- I have found some instances for Assignment 1 where students are clearly using ChatGPT to do their work.
- These students will be receiving an email from the Office of Community Standards in the next few weeks or so.
- When those cases are complete, and if the students are found responsible, the students in question will fail the class.
- If you have anything to tell me in regard to your use of ChatGPT or otherwise, please email me at cgregg@stanford.edu
- I take this very seriously, because I want the honest students to feel secure that they are doing the right thing.
The String Library: strcmp and strncmp

**strcmp**: Compares two strings, character-by-character, and returns 0 for identical strings, < 0 if s is before t in the alphabet, and > 0 if s is after t (digits are less than alphabetic characters). Prototype:

```c
int strcmp(const char *s, const char *t);
```

**strncmp**: Performs the same comparison as strcmp except that it stops after n characters (and does not traverse past null characters). Prototype:

```c
int strncmp(const char *s, const char *t, size_t n);
```
The String Library: strcmp and strncmp

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(int argc, char **argv)
{
    char *s1 = argv[1];
    char *s2 = argv[2];
    int cmplen = atoi(argv[3]);

    int cmp_result = strcmp(s1, s2);

    char *result_text;
    if (cmp_result == 0) {
        result_text = "is the same as";
    } else if (cmp_result < 0) {
        result_text = "comes before";
    } else {
        result_text = "comes after";
    }

    printf("String "%s" %s "%s" in the alphabet.
", s1, result_text, s2);

    return 0;
}
```

```
$ ./strcmp_ex cat camel 2
String "cat" comes after "camel" in the alphabet.
Up to character 2, "cat" is the same as "camel" in the alphabet.
```
The String Library: strchr

`strchr`: Returns a pointer to the first occurrence of a character in s, or NULL if the character is not in the string. Prototype:

```c
char *strchr(const char *s, int ch);
```

Example:

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(int argc, char **argv)
{
    char *word = argv[1];
    char ch = argv[2][0];

    printf("\"%s\" pointer: %p\n", word, word);
    printf("pointer to the first instance of %c in %s: %p\n", ch, word, strchr(word, ch));
    return 0;
}
```

```
$ ./strchr_ex fabulous u
"fabulous" pointer: 0x7ffee9c888c4
pointer to the first instance of u in fabulous: 0x7ffee9c888c7
```

```
$ ./strchr_ex fabulous r
"fabulous" pointer: 0x7ffee0c328c4
pointer to the first instance of r in fabulous: (nil)
```
**strstr**: Locate a substring. Returns a pointer to the first occurrence of needle in haystack, or NULL if the substring does not exist.

```c
char *strstr(const char *haystack, const char *needle);
```

```c
// file: strstr_ex.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(int argc, char **argv)
{
    char *haystack = argv[1];
    char *needle = argv[2];

    printf("\"%s\" pointer: %p\n", haystack, haystack);
    printf("pointer to the first instance of \"%s\" in %s: %p\n", 
           needle, haystack, strstr(haystack, needle));
    return 0;
}
```

$ ./strstr_ex mississippi ssip
"mississippi" pointer: 0x7ffeeb06b8bc
pointer to the first instance of "ssip" in mississippi: 0x7ffeeb06b8c1
**strcpy**: Copies `src` to `dst`, including the null byte. The caller is responsible for ensuring that there is enough space in `dst` to hold the entire copy. The strings may not overlap.

```c
char *strcpy(char *dst, const char *src);
```

```c
// file: strcpy_ex.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(int argc, char **argv)
{
    char *word = argv[1];
    // +1 necessary below for terminating null byte
    char wordcopy[strlen(word) + 1];

    strcpy(wordcopy, word);
    word[0] = 'x';
    wordcopy[0] = 'y';

    printf("word: %s\n", word);
    printf("wordcopy: %s\n", wordcopy);
    return 0;
}
```

Be careful! The **strcpy** function is responsible for many "buffer overflows" where the destination did not have enough space for the source! This is where nefarious hackers do their thing!
**strncpy**: Similar to strcpy, except that at most n bytes will be copied. If there is no null byte in the first n bytes of src, then dst will not be null-terminated!

```c
char *strncpy(char *dst, const char *src, size_t n);
```

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

const int MAX_WORDLEN = 5;

int main(int argc, char **argv) {
    char *word = argv[1];
    char wordcopy[MAX_WORDLEN];

    // only copy up to one before the end
    strncpy(wordcopy, word, MAX_WORDLEN - 1);
    // put a null at the end in case the word is too long
    wordcopy[MAX_WORDLEN - 1] = '\0';

    printf("word: %s\n", word);
    printf("wordcopy: %s\n", wordcopy);
    return 0;
}
```

Again, be careful! The **strncpy** function won't put a null at the end of the copy automatically!
The String Library: strncpy

The following is a buggy version, without the appropriate checks!

// file: strncpy_buggy.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
const int MAX_WORDLEN = 5;

int main(int argc, char **argv) {
    char *word = argv[1];
    char wordcopy[MAX_WORDLEN];
    strncpy(wordcopy, word, MAX_WORDLEN);
    printf("word: %s\n", word);
    printf("wordcopy: %s\n", wordcopy);
    return 0;
}

$ ./strncpy_buggy wonderful
word: wonderful
wordcopy: wonde????

This program has a buffer overflow! Five chars were copied, but it doesn't put on the necessary null. This is bad code, and gets people fired from their jobs.
**The String Library: strcat and strncat**

**strcat** and **strncat**: "Concatenate" two strings by appending `src` onto the end of `dst`. **strncat** only copies up to `n` bytes, and `dst` is always null-terminated, which adds an extra byte!

```c
char *strcat(char *dst, const char *src);
char *strncat(char *dst, const char *src, size_t n);
```

Be careful -- you have to determine the size of the buffer to copy into, and it takes a bit of arithmetic, especially in the case of **strncat**. If you are trying to create space that is exactly the right size, use **man strncat** to read up to refresh your memory.
The String Library: strcat and strncat

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

const int MAX_CPY = 3;

int main(int argc, char **argv)
{
    char *word1 = argv[1];
    char *word2 = argv[2];
    size_t total_len = strlen(word1) + strlen(word2);
    // word1cpy_a will hold word1 + word 2,
    // so we need an extra byte
    char word1cpy_a[total_len + 1];
    // word1cpy_b will hold word1 + 3 bytes of word2,
    // and we need an extra byte for the null
    char word1cpy_b[strlen(word1) + MAX_CPY + 1];
    strcpy(word1cpy_a, word1);
    strcpy(word1cpy_b, word1);
    strcat(word1cpy_a, word2);
    strncat(word1cpy_b, word2, MAX_CPY);
    printf("%s + %s = %s\n", word1, word2, word1cpy_a);
    printf("%s + first %d bytes of %s = %s\n",
           word1, MAX_CPY, word2, word1cpy_b);
    return 0;
}
```

How many bytes does "happybirthday" require?

```
$ ./strcat_ex happy birthday
happy + birthday = happybirthday
happy + first 3 bytes of birthday = happybir
```

16
The String Library: `strcat` and `strncat`

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

const int MAX_CPY = 3;

int main(int argc, char **argv)
{
    char *word1 = argv[1];
    char *word2 = argv[2];

    size_t total_len = strlen(word1) + strlen(word2);
    // word1cpy_a will hold word1 + word2,
    // so we need an extra byte
    char word1cpy_a[total_len + 1];

    // word1cpy_b will hold word1 + 3 bytes of word2,
    // and we need an extra byte for the null
    char word1cpy_b[strlen(word1) + MAX_CPY + 1];

    strcpy(word1cpy_a, word1);
    strcpy(word1cpy_b, word1);
    strcat(word1cpy_a, word2);
    strncat(word1cpy_b, word2, MAX_CPY);

    printf("%s + %s = %s\n", word1, word2, word1cpy_a);
    printf("%s + first %d bytes of %s = %s\n", word1, MAX_CPY, word2, word1cpy_b);

    return 0;
}
```

$ ./strcat_ex happy birthday
happy + birthday = happybirthday
happy + first 3 bytes of birthday = happybir

How many bytes does "happybirthday" require? 14

(5 for happy, 8 for birthday, 1 for null)

```c
strlen("happy") == 5
strlen("birthday") == 8
```

So, we need 5 + 8 + 1 = 14
The String Library: strcat and strncat

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

const int MAX_CPY = 3;

int main(int argc, char **argv)
{
    char *word1 = argv[1];
    char *word2 = argv[2];

    size_t total_len = strlen(word1) + strlen(word2);
    // word1cpy_a will hold word1 + word 2,
    // so we need an extra byte
    char word1cpy_a[total_len + 1];

    // word1cpy_b will hold word1 + 3 bytes of word2,
    // and we need an extra byte for the null
    char word1cpy_b[strlen(word1) + MAX_CPY + 1];

    strcpy(word1cpy_a, word1);
    strcpy(word1cpy_b, word1);
    strcat(word1cpy_a, word2);
    strncat(word1cpy_b, word2, MAX_CPY);

    printf("%s + %s = %s\n", word1, word2, word1cpy_a);
    printf("%s + first %d bytes of %s = %s\n", word1, MAX_CPY, word2, word1cpy_b);

    return 0;
}
```

$ ./strcat_ex happy birthday
happy + birthday = happybirthday
happy + first 3 bytes of birthday = happybir

How many bytes does "happybirthday" require? 14
(5 for happy, 8 for birthday, 1 for null)

How many bytes does "happybir" require?
```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

const int MAX_CPY = 3;

int main(int argc, char **argv)
{
    char *word1 = argv[1];
    char *word2 = argv[2];

    size_t total_len = strlen(word1) + strlen(word2);
    // word1cpy_a will hold word1 + word 2,
    // so we need an extra byte
    char word1cpy_a[total_len + 1];

    // word1cpy_b will hold word1 + 3 bytes of word2,
    // and we need an extra byte for the null
    char word1cpy_b[strlen(word1) + MAX_CPY + 1];

    strcpy(word1cpy_a, word1);
    strcpy(word1cpy_b, word1);
    strcat(word1cpy_a, word2);
    strncat(word1cpy_b, word2, MAX_CPY);

    printf("%s + %s = %s\n", word1, word2, word1cpy_a);
    printf("%s + first %d bytes of %s = %s\n", word1, MAX_CPY, word2, word1cpy_b);

    return 0;
}
```

$ ./strcat_ex happy birthday
happy + birthday = happybirthday
happy + first 3 bytes of birthday = happybir

How many bytes does "happybirthday" require? 14
(5 for happy, 8 for birthday, 1 for null)

How many bytes does "happybir" require? 9
(5 for happy, 3 for bir, 1 for null)

`strlen("happy") = 5`

We will copy at most 3 bytes from word2.
We need 5 + 3 + 1 for the total with null.
The String Library: strspn

**strspn**: Calculates and returns the length in bytes of the initial part of `s` which contains only characters in `accept`.

For example, `strspn("hello", "efgh")` returns 2 because only the first two characters in “hello” are in “efgh.”

```
size_t strspn(const char *s, const char *accept)
```

Learn this function well! It tends to make an appearance on CS 107 midterms and finals!
**The String Library: strcspn**

`strcspn`: Similar to `strspn` except that `strcspn` returns the length in bytes of the initial part of `s` which **does not** contain any characters in `reject`.

For example, `strcspn("hello", "mnop")` returns 4 because the first four characters in “hello” are not in “mnop.”

```
size_t strcspn(const char *s, const char *reject);
```

Learn this function well, and make sure you understand how it works and the difference between `strspn` and `strcspn`!

BTW, the "c" in `strcspn` stands for "complement" -- the complement of the reject characters is what is being "spanned".
// file: strspn_ex.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(int argc, char **argv)
{
    char *word = argv[1];
    char *charset_accept = argv[2];
    char *charset_reject = argv[3];

    size_t strspn_count = strspn(word, charset_accept);
    size_t strcspn_count = strcspn(word, charset_reject);

    printf("The first %lu initial characters in \"%s\" are in \"%s\"\n", strspn_count, word, charset_accept);
    printf("The first %lu initial characters in \"%s\" are not in \"%s\"\n", strcspn_count, word, charset_reject);
    return 0;
}

$ ./strspn_ex tremendous rtme dmns
The first 5 initial characters in "tremendous" are in "rtme"
The first 3 initial characters in "tremendous" are not in "dmns"
**The String Library: strdup and strndup**

**strdup**: Returns a pointer to a *heap-allocated* string which is a copy of s. It is the responsibility of the caller to free the pointer when it is no longer needed.

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

**strndup**: Like strdup but only copies up to n bytes. The resulting string will be null-terminated.

```c
char *strndup(const char *s, size_t n);
```

These two functions take care of allocating space for the duplicate of the string, but both require the **calling function** to *free* the copy when it is no longer needed. If the copy isn't freed, this is considered a *memory leak*, and can waste memory.
The String Library: strdup and strndup

// file: strdup_ex.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

const int BYTES_TO_COPY = 3;

int main(int argc, char **argv)
{
    char *word = argv[1];

    // remember to free these!
    char *word_copy = strdup(word);
    char *word_copy3 = strndup(word, BYTES_TO_COPY);

    printf("word: %s\n", word);
    printf("word_copy: %s\n", word_copy);
    printf("First %d letters of word: %s\n", BYTES_TO_COPY, word_copy3);

    // free the memory once no longer needed
    free(word_copy);
    free(word_copy3);

    return 0;
}
Why don't strings keep their own length?

C strings differ from C++ strings in that they are simple, and are just a null-terminated character array.

Strings didn't have to be this way -- when C was being developed, another popular language, Pascal, had "length-prefix" strings, which stored the length in the first byte of the string. Although this made finding the length of a string $O(1)$, it limited the size of strings to 256 characters! (Later versions of Pascal added support for up to 64-bit prefixes, but this had the downside of adding length to the string, which takes up space).

The original justification in C was that having only 1-byte of overhead was nice because memory was limited (remember this was the 1970s!), and the terminating null was better than a prefix-byte because it didn't limit the size of the string.
References and Advanced Reading

• References:
  • https://www.tutorialspoint.com/c_standard_library/ctype_h.htm
  • https://www.tutorialspoint.com/c_standard_library/string_h.htm

• Advanced Reading:
  • https://www.cs.bu.edu/teaching/cpp/string/array-vs-ptr/
  • What is the justification for a null-terminated string? https://stackoverflow.com/questions/4418708/whats-the-rationale-for-null-terminated-strings
  • Interesting criticism of the Pascal language for its string type: http://www.lysator.liu.se/c/bwk-on-pascal.html