

CS 107

Lecture 4: Chars and C-Strings

Friday, January 14, 2022

Computer Systems
Winter 2022
Stanford University
Computer Science Department

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.

Lecturers: Chris Gregg

s	t	r	i	n	g	\0
----------	----------	----------	----------	----------	----------	-----------



Today's Topics

- Logistics
 - Assign1 — Due Wednesday at 11:59pm, grace period until Friday.
 - Feedback: during the quarter you should receive a couple of feedback emails about the course. Please be honest — I want to improve the course where necessary! Constructive feedback and criticism is always appreciated.
- 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



C's char type



C's `char` type

Most likely, you are already familiar with the `char` type from other courses. In C, `chars` are defined to be a 1-byte value, and most often `chars` are signed, although we usually only use 0-127 for character data (see below).

A `char` does not necessarily have to hold alphabetic or numeric character data, but often it does, and in C, the ASCII character set defines the encoding between the numeric value of the `char` and its character mapping. We will limit ourselves to character data in the range of 0 - 127, which is what ASCII defines.

There is a standard called "unicode" that you will investigate for Assignment 1, but for CS 107, we will limit ourselves to the ASCII character set.



The ctype library

One of the standard libraries you should become familiar with is the "ctype" library, which includes many functions that act on character data.

The functions usually take an `int` instead of a `char`, and this is because the functions can accept the full unsigned char range (0 - 255) plus the special character `EOF` ("end of file"), which is often represented by -1.

We can see information about the ctype functions by typing `man function`, where function is one of the following (there are more, but we only care about these):

`isalpha`, `isdigit`, `isalnum`, `islower`, `isupper`, `isspace`, `isxdigit`, `tolower`, and `toupper`. You can get a list of most of them with a combination of `man isalpha` and `man tolower`.



The ctype library

The following code demonstrates some of the functions in the ctype library:

```
// file: ctypedemo.c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

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

    // count alpha characters, digits,
    // whitespace, and punctuation
    int alphacount = 0;
    int digitcount = 0;
    int spacecount = 0;
    int punctcount = 0;
    int total = 0;
    int i = 0;
```

```
...

    while (string[i] != 0) {
        if (isalpha(string[i])) alphacount++;
        if (isdigit(string[i])) digitcount++;
        if (isspace(string[i])) spacecount++;
        if (ispunct(string[i])) punctcount++;
        total++;
        i++;
    }
    printf("Alphabetic characters: %d\n", alphacount);
    printf("Digits: %d\n", digitcount);
    printf("Spaces: %d\n", spacecount);
    printf("Punctuation: %d\n", punctcount);
    printf("Total characters: %d\n", total);

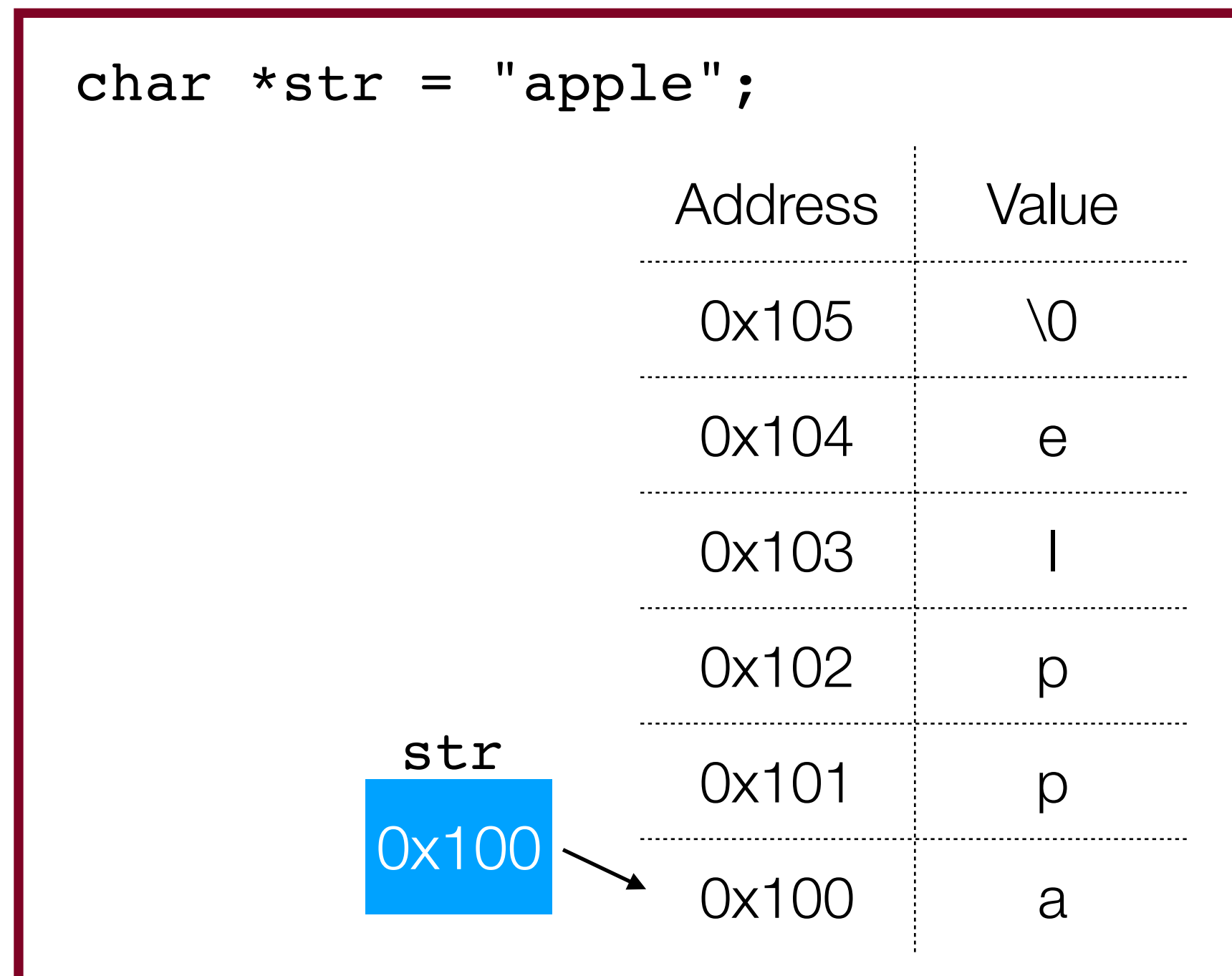
    return 0;
}
```



C Strings

C strings are simply a sequence of `chars`, followed by a terminating 0 (called a "null" byte).

C strings are referenced by a *pointer* to its first character, or by an array variable, which is converted to a pointer when we need to access the elements:



Let's take a moment to look at this diagram -- we will see many like it during the quarter.

1. `str` is a variable that holds the address of the first character in "apple".
2. We have drawn the array vertically, with the lowest address at the bottom
3. Each character is 1 byte away from the previous character.



C Strings

It is meaningless in C to compare strings by their pointer values:

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

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

    // the following two lines do not compare
    // the two strings!
    if (s1 < s2) printf("%s is less than %s\n",s1,s2);
    if (s1 == s2) printf("%s is equal to %s\n",s1,s2);
    if (s1 > s2) printf("%s is greater than %s\n",s1,s2);

    printf("%s address: %p\n",s1,s1);
    printf("%s address: %p\n",s2,s2);

    return 0;
}
```



```
$ gcc -g -O0 -std=gnu99 -Wall
    pointer_compare.c -o pointer_compare

$ ./pointer_compare cat dog
cat is less than dog
cat address: 0x7fffeef0e9962
dog address: 0x7fffeef0e9966
$ ./pointer_compare dog cat
dog is less than cat
dog address: 0x7fffeeb6b7962
cat address: 0x7fffeeb6b7966
```

Wrong!



C Strings

Assigning a string pointer to another string pointer does not make a copy of the original string! Instead, both pointers point to the **same** string.

Because of this, changing a character via either pointer changes the string.

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

int main(int argc, char **argv)
{
    char *s1 = argv[1];
    char *s2 = s1; // not a copy!

    s1[0] = 'x';
    s2[1] = 'y';

    printf("address: %p, string:%s\n",s1,s1);
    printf("address: %p, string:%s\n",s2,s2);

    return 0;
}
```



```
$ gcc -g -O0 -std=gnu99 -Wall string_pointers.c
-o string_pointers $ ./string_pointers cs107
$ ./string_pointers cs107
address: 0x7ffee837f962, string:xy107
address: 0x7ffee837f962, string:xy107
```



The String Library

One of the more important libraries for CS 107 is the string library, `<string.h>`

You need to be *very* familiar with the library functions we will discuss, and you may see any of them on the midterm and final exams.

Do not re-write these functions unless asked to explicitly for an assignment! The string library is finely tuned, and it works. It isn't worth the time or effort to try and re-write the string library functions (and we will take points off if you do!)

String library functions all have a worst-case complexity of $O(n)$. This is because strings are *not* objects, and don't have any information (e.g., the string length) embedded in them.



The String Library: strlen

strlen: Calculates and returns the length of the string. Prototype:

```
size_t strlen(const char *str);
```

Example:

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

int main(int argc, char **argv)
{
    printf("argv[1], \"%s\", has a length of %zu characters.\n",
           argv[1], strlen(argv[1]));
    return 0;
}
```



Want a challenge?
See musl's version of
strlen: <https://github.com/esmil/musl/blob/master/src/string/strlen.c>

```
$ ./strlen_ex cs107
argv[1], "cs107", has a length of 5 characters.
```



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:

```
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:

```
int strncmp(const char *s, const char *t, size_t n);
```



The String Library: strcmp and strncmp

```
// file: strcmp_ex.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.\n",
        s1,result_text,s2);
}
```

```
cmp_result = strncmp(s1,s2,cmplen);
    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("Up to character %d, \"%s\" %s \"%s\" in the alphabet.\n",
        cmplen,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:

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

```
// file: strchr_ex.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("\n%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)
```



The String Library: strstr


strstr: Locate a substring. Returns a pointer to the first occurrence of needle in haystack, or NULL if the substring does not exist.

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

```
// 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
```



3 minute break



The String Library: strcpy

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.


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

```
// 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;
}
```



```
$ ./strcpy_ex hello
word: xello
wordcopy: yello
```

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!



The String Library: strncpy

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!

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

```
// file: strncpy_ex.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;
}
```



```
$ ./strncpy_ex wonderful
word: wonderful
wordcopy: wond
```

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[?][?]J[?][?][?]
```

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!

```
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



```
// file: strcat_ex.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?



The String Library: strcat and strncat

```
// file: strcat_ex.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)


`strlen("happy") == 5`

`strlen("birthday") == 8`

So, we need $5 + 8 + 1 = 14$



The String Library: strcat and strncat



```
// file: strcat_ex.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?



The String Library: strcat and strncat

```
// file: strcat_ex.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”.



The String Library: strspn and strcspn example

```
// 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.:

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

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

```
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;
}
```



```
$ ./strdup_ex February
word: February
word_copy: February
First 3 letters of word: Feb
```



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-prefixed" 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://en.wikibooks.org/wiki/C_Programming/String_manipulation
- 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/>
- <https://www.quora.com/Why-dont-we-need-null-character-in-arrays-as-in-strings-to-know-its-end-point>
- 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>

