CS107 Winter 2020, Lecture 5
More C Strings

reading:
Reading: K&R (1.9, 5.5, Appendix B3) or Essential C section 3

Lisa helper hours for this lecture’s content: Tuesday 1/21, 5-7pm

Slides by Jerry Cain and Lisa Yan, who leveraged prior work by Nick Troccoli, Julie Zelenski, Marty Stepp, Cynthia Lee, Chris Gregg, and others.
Plan For Today

- **Recap**: Strings and pointers
- More common string operations: Concatenating, searching, and spans
- **Break**: Announcements
- Strings, memory, and pointers, part 2
- Double pointers and arrays of strings
Plan For Today

• **Recap**: Strings and pointers
  • More common string operations: Concatenating, searching, and spans

• **Break**: Announcements
  • Strings, memory, and pointers, part 2
  • Double pointers and arrays of strings
Key takeaways from last time

1. Valid strings are null-terminated.

```plaintext
char str[] = "Hello";
int length = strlen(str); // 5
```
Key takeaways from last time

1. Valid strings are null-terminated.

2. An array name (and a string name, by extension) is the address of the first element.

```
char str[] = "Hello";
int length = strlen(str);  // 5
char *ptr = str+1;          // 0xf1
printf("%s\n", ptr);        // prints "ello"
```
1. Valid strings are null-terminated.
2. An array name (and a string name, by extension) is the address of the first element.
3. All parameters in C are “pass by value.” For efficiency purposes, arrays (and strings, by extension) passed in as parameters are converted to pointers.
Code study: `string.h` implementations

```c
#include <string.h>

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

`src`: A modifiable pointer to `(const char)`
Code study: `string.h` implementations

```c
char *my_strcpy(char *dst, const char *src)
{
    char *result = dst;
    while (*dst++ = *src++);
    return result;
}
```

Key takeaways:

• While loop with no body executes until zero condition

⚠ The postfix ++ operator increments the value of a variable after execution.

• Assignment op ( = ) returns result of assignment
Code study: string.h implementations

```c
1 char *my_strcpy(char *dst, const char *src)
2 {
3     char *result = dst;
4     while ((*dst++ = *src++));
5     return result;
6 }
```

Key takeaways:
- While loop with no body executes until zero condition
- The postfix ++ operator increments the value of a variable after execution.
- Assignment op (=) returns result of assignment
• Write a function `diamond` that accepts a string parameter and prints its letters in a "diamond" format as shown below.
  • For example, `diamond("PICHU")` should print:

```
P
PI
PIC
PICH
PICHU
ICHU
CHU
HU
U
```
Practice: Diamond

```bash
cp -r /afs/ir/class/cs107/samples/lectures/lect5 .
```
Plan For Today

• **Recap**: Strings and pointers
• **More common string operations**: Concatenating, searching, and spans
• **Break**: Announcements
  • Strings, memory, and pointers, part 2
  • Double pointers and arrays of strings
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strlen((str))</td>
<td>returns the # of chars in a C string (before null-terminating character).</td>
</tr>
<tr>
<td>strcmp((str1, str2)), strncmp((str1, str2, n))</td>
<td>compares two strings; returns 0 if identical, &lt;0 if (str1) comes before (str2) in alphabet, &gt;0 if (str1) comes after (str2) in alphabet. (strncpy) stops comparing after at most (n) characters.</td>
</tr>
<tr>
<td>strchr((str, ch)), strrchr((str, ch))</td>
<td>character search: returns a pointer to the first occurrence of (ch) in (str), or NULL if (ch) was not found in (str). (strrchr) finds the last occurrence.</td>
</tr>
<tr>
<td>strstr((haystack, needle))</td>
<td>string search: returns a pointer to the start of the first occurrence of (needle) in (haystack), or NULL if (needle) was not found in (haystack).</td>
</tr>
<tr>
<td>strcpy((dst, src)), strncpy((dst, src, n))</td>
<td>copies characters in (src) to (dst), including null-terminating character. Assumes enough space in (dst). Strings must not overlap. (strncpy) stops after at most (n) chars (might not include null-terminating character).</td>
</tr>
<tr>
<td>strcat((dst, src)), strncat((dst, src, n))</td>
<td>concatenate (src) onto the end of (dst). (strncat) stops concatenating after at most (n) characters. Always adds a null-terminating character.</td>
</tr>
<tr>
<td>strspn((str, accept)), strcspn((str, reject))</td>
<td>(strspn) returns the length of the initial part of (str) which contains only characters in (accept). (strcspn) returns the length of the initial part of (str) which does not contain any characters in (reject).</td>
</tr>
</tbody>
</table>
Concatenating Strings

What does the following code do?

```c
char str1[] = "tomato";
char str2[] = "potato";
printf("%s", str1 + str2);
```

A. Prints "tomatopotato", 12 characters
B. Prints "tomato\0potato", 13 characters
C. Prints sum of two memory addresses
D. Compiler error
What does the following code do?

```c
char str1[] = "tomato";
char str2[] = "potato";
printf("%s", str1 + str2);
```

A. Prints "tomatopotato", 12 characters
B. Prints "tomato\0potato", 13 characters
C. Prints sum of two memory addresses
D. Compiler error

⚠ Pointer arithmetic: You cannot add addresses together (result will most likely refer to inaccessible memory)
The string library: strcat

To concatenate strings, use strcat (or strncat) which will both remove the old '\0' and add a new one at the end.

```c
1 char str1[13]; // enough space for strings + '\0'
2 strcpy(str1, "hello ");
3 char str2[] = "world!";
4 strcat(str1, str2);
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>h</td>
<td>e</td>
<td>l</td>
<td>l</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>o</td>
<td>r</td>
<td>l</td>
<td>d</td>
<td>!</td>
<td>\0</td>
</tr>
</tbody>
</table>

str1: 'h' 'e' 'l' 'l' 'o' ' ' '\0' ? ? ? ? ? ? ?

str2: 'w' 'o' 'r' 'l' 'd' '!' '\0'
The string library: strcat

To concatenate strings, use strcat (or strncat) which will both remove the old '\0' and add a new one at the end.

```
1 char str1[13]; // enough space for strings + '\0'
2 strcpy(str1, "hello ");
3 char str2[] = "world!";
4 strcat(str1, str2);
```

```
```
String Spans

`strspn(str, accept):` returns the *length* of the initial part of `str` which contains **only** characters in `accept`.

```c
char str[] = "Pocket Monsters";
int span = strspn(str, "Pokemon"); // 2
```
strspn(str, accept): returns the length of the initial part of str which contains only characters in accept.

```
char str[] = "Pocket Monsters";
int span = strspn(str, "Pokemon");   // 2
```

strcspn(str, reject): returns the length of the initial part of str which contains only characters not in reject (strcspn: ”complement”).

```
char str[] = "Pocket Monsters";
int span = strcspn(str, "QqJjZzXxVvKk");  // 3
```
Practice: Pig Latin

pig.c

string.h functions to consider: strcspn, strcat, strncat
Plan For Today

- **Recap**: Strings and pointers
- More common string operations: Concatenating, searching, and spans
- **Break**: Announcements
- Strings, memory, and pointers, part 2
- Double pointers and arrays of strings
Announcements

• Assignment 0 grades released latest Monday

• Assignment 1 due Monday 11:59PM PST
  • **Grace period** until Wed. 11:59PM PST

• Lab 2: C strings practice

• Assignment 2 released at Assignment 1 due date
  • Due next Mon. 11:59PM PST, grace period until next Wed. 11:59PM PST
  • Programs using C strings
Joke break

https://xkcd.com/138/
Plan For Today

• Recap: Strings and pointers
• More common string operations: Concatenating, searching, and spans
• Break: Announcements

• Strings, memory, and pointers, part 2
• Double pointers and arrays of strings
Important details: char * vs char []

• As parameter types, they are both pointers (pass by value + efficiency):

```c
void fun_times(char *str);      // as a parameter
void fun_times(char str[]);    // as a parameter
```

• However...

⚠ Arrays are **NOT** pointers, even if they sometimes behave like them.

Always keep in mind the following when working with arrays and pointers:
• Can we change the memory address?
• Can we change the content at this memory address?
• **Arrays** are instantiated as a contiguous block of memory on the stack.

```
// chars stored in // stack frame
char arr[] = "hi";
```

• The array name is the address of the first element, designated at compile-time. It refers to the original block of memory.

• You can never change the address of an array, but you can always modify its contents.*

*ignoring const and/or static arrays
• A **pointer** is a variable that stores a memory address.
• This memory can be anywhere: in the stack, heap, or the data segment.
• The **data segment** contains global/static variables or read-only **string literals**.

```c
// chars stored in read-only DS
char *str = "hi";

// chars stored in stack frame
char arr[] = "hi";
char *str = arr;
```

• You can always change the address stored in a pointer, even if you might not be able to modify the content at that address.
**char* vs char[] exercises**

Suppose we use a variable `str` as follows:

```c
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

- Will there be a compile error/segfault?
- If no errors, what is printed?

1. `char str[6] = "Hello1";`
2. `char *str = "Hello2";`
3. `char arr[] = "Hello3"; char *str = arr;`
4. `char *ptr = "Hello4"; char *str = ptr;`
Suppose we use a variable `str` as follows:

```
str = str + 1;
str[1] = 'u';
printf("%s", str)
```

For each of the following instantiations:

1. `char str[6] = "Hello1";`
   - Compile error (cannot reassign array)

2. `char *str = "Hello2";`
   - Segmentation fault (string literal)

3. `char arr[] = "Hello3"; char *str = arr;`
   - Prints `eu1o3`

4. `char *ptr = "Hello4"; char *str = ptr;`
   - Segmentation fault (string literal)
Nitty gritty detail: char * vs char[]

⚠ Detail #1: sizeof() takes the size of a variable at compile time.

void binky(char arr[]);
int main(int argc, char *argv[]) {
    char arr[] = "supercalifragilisticexpialidocious";
    printf("sizeof: %ld\n", sizeof(arr));          // sizeof: 35
    binky(arr);
    ...
}
void binky(char arr[]) {                          // arr is pointer
    printf("sizeof: %ld\n", sizeof(arr));         // sizeof: 8
    printf("strlen: %ld\n", strlen(arr));         // strlen: 34
}

Use strlen instead of sizeof, or pass in array length as parameter.
Detail #2: Array initialization will **always** instantiate memory on the stack.

- `char *ptr = "hi";`
  - Initialize string literal, stored in data segment
  - Point local variable `ptr` to address of first character in string literal

- `char arr[] = "hi";`
  - Allocate 3 chars’ worth of stack space, where `arr` points to address of first element
  - Set elements of `arr` to 'h', 'i', and '\0'

Treat as “fun fact” for now; you will be able to verify this with assembly in a few weeks 😊
What’s the deal with pointers?

Why even bother with pointers if arrays can (seemingly) do most of the work?

• Pointers allow us to (effectively) pass by reference.
• Pointers are always 8* bytes, so they can refer to large data structures in a compact way.
• Pointers let us refer to memory anywhere (not just on the stack).
• Pointers to pointers are really useful.

*on a 64-bit machine like myth
Plan For Today

• Recap: Strings and pointers
• More common string operations: Concatenating, searching, and spans
• Break: Announcements
• Strings, memory, and pointers, part 2

• Double pointers and arrays of strings
We can make an array of strings to group multiple strings together:

```c
char *string_array[5]; // array of 5 char *s
```

We can also use the following shorthand to initialize a string array:

```c
char *string_array[] = {
    "hello world",
    "hi",
    "have a nice day",
};
```
argc and argv

```
int main(int argc, char *argv[]);
```

# elements in argv

An array of char *’s!!
Double pointer parameters

The parameter types below are equivalent (both are double pointers):

```c
void binky(char **x);
void winky(char ***x);
```

(suggestion): want to process an array of pointers  
(suggestion): want to modify a pointer’s address

• As a C programmer, you often choose stylistically between the two to convey meaning, based on what you expect the input to be.
• You should feel free to use your own abstraction.
Write a function **skip_spaces** that modifies a string pointer to skip past any leading spaces.

```c
void skip_spaces(__?__) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skip_spaces(__?__);
    printf("%s", str); // should print "hello"
}
```
Write a function `skip_spaces` that modifies a string pointer to skip past any leading spaces.

```c
void skip_spaces(char **strptr) {
    ...
}

int main(int argc, char *argv[]) {
    char *str = "    hello";
    skip_spaces(&str);
    printf("%s", str);    // should print "hello"
}
```

We are modifying a specific instance of the string pointer, so we pass the `location` of the string pointer we would like to modify.
Demo: Skip spaces

`skip_spaces.c`
Write a function `verify_password` that accepts a candidate password and certain password criteria and returns whether the password is valid.

```c
bool verify_password(char *password, char *bad_substrings[], int num_bad_substrings);
```

password is **valid** if it does not contain any substrings in `bad_substrings`. 
Demo: Password Verification

verify_password.c
Recap of today

- **Recap**: Strings and pointers
- More common string operations: Concatenating, searching, and spans
- **Break**: Announcements
- Strings, memory, and pointers, part 2
- Double pointers and arrays of strings