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

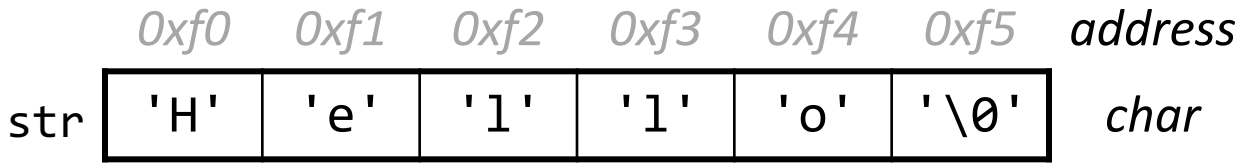
- **Recap**: Strings and pointers
- More common string operations: Concatenating, searching, and spans
- **Break**: Announcements
- Strings, memory, and pointers, part 2
- Double pointers and arrays of strings

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

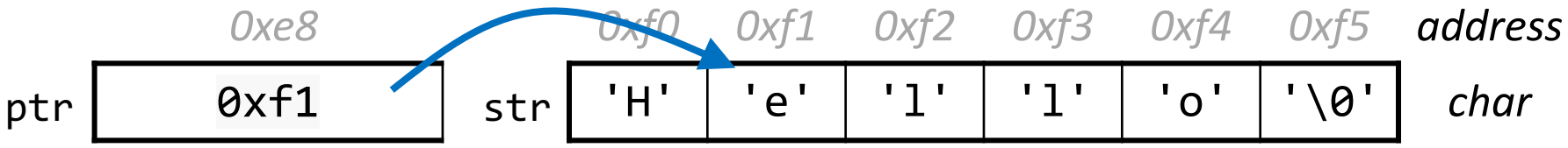
1. Valid strings are null-terminated.




```
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"
```

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.
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

```
STRCPY(3)                  Linux Programmer's Manual                  STRCPY(3)

NAME
       strcpy, strncpy - copy a string

SYNOPSIS
       #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

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

string_code.c

- Line 4: What is happening?

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

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

```
*dst = *src;
char c = *dst;
src = src + 1;
dst = dst + 1;
// break if c == '\0'
```

string_code.c

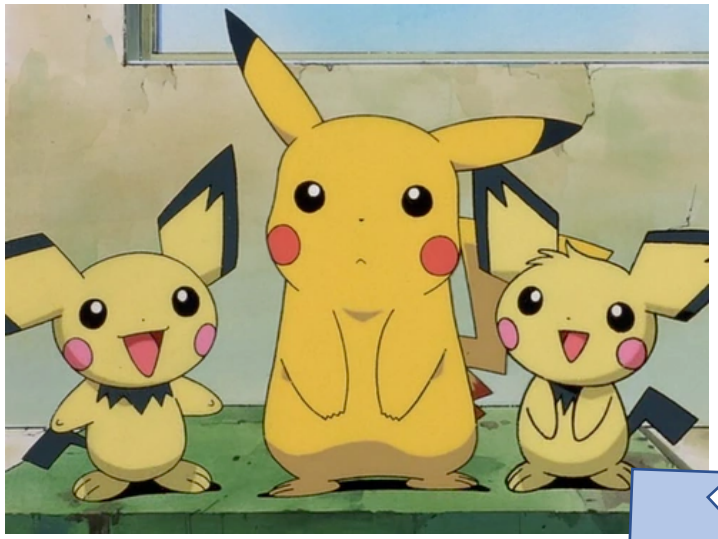
Key takeaways:

- 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

String Diamond

- 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

```
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

Common `string.h` Functions

Function	Description
`strlen(`***`str`***`)`	returns the # of chars in a C string (before null-terminating character).
`strcmp(`***`str1, str2`***`),` `strncmp(`***`str1, str2, n`***`)`	compares two strings; returns 0 if identical, <0 if ***`str1`*** comes before ***`str2`*** in alphabet, >0 if ***`str1`*** comes after ***`str2`*** in alphabet. ***`strncmp`*** stops comparing after at most ***`n`*** characters.
`strchr(`***`str, ch`***`)` `strrchr(`***`str, ch`***`)`	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.
`strstr(`***`haystack, needle`***`)`	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`***.
`strcpy(`***`dst, src`***`),` `strncpy(`***`dst, src, n`***`)`	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).
`strcat(`***`dst, src`***`),` `strncat(`***`dst, src, n`***`)`	concatenate ***`src`*** onto the end of ***`dst`***. **`strncat`** stops concatenating after at most ***`n`*** characters. Always adds a null-terminating character.
`strspn(`***`str, accept`***`),` `strcspn(`***`str, reject`***`)`	**`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`***.

Concatenating Strings

What does the following code do?

```
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

Concatenating Strings

What does the following code do?

```
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

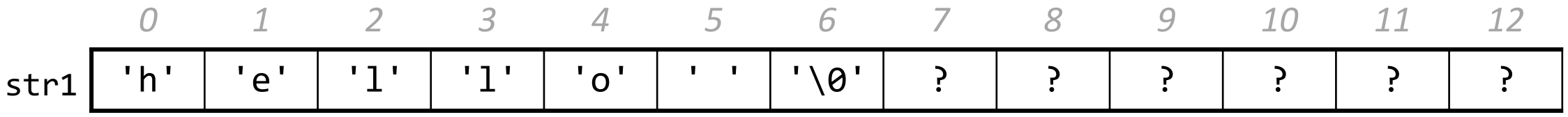
```
error: invalid operands to binary
+ (have ‘char *’ and ‘char *’)
     printf("%s", str1 + str2);
```

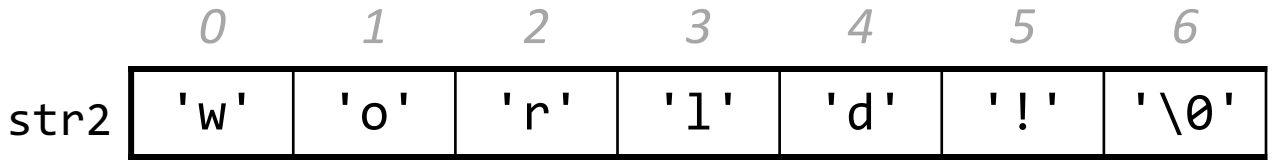
⚠ Pointer arithmetic: You cannot add addresses together (result will most likely refer to inaccessible memory)

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.

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

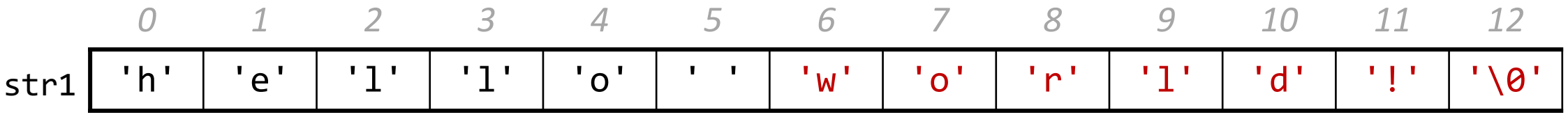


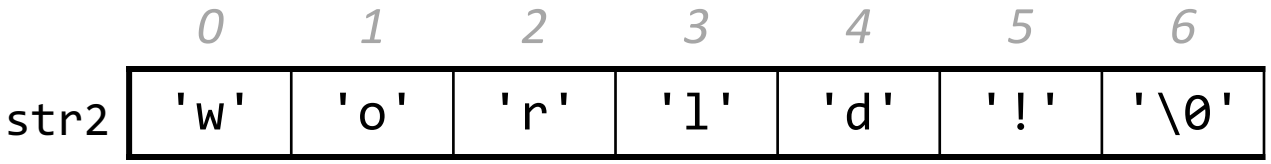


The string library: strcat

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`.

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

String Spans

`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):

```
void fun_times(char *str);
```

```
void fun_times(char str[]);
```

- 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?

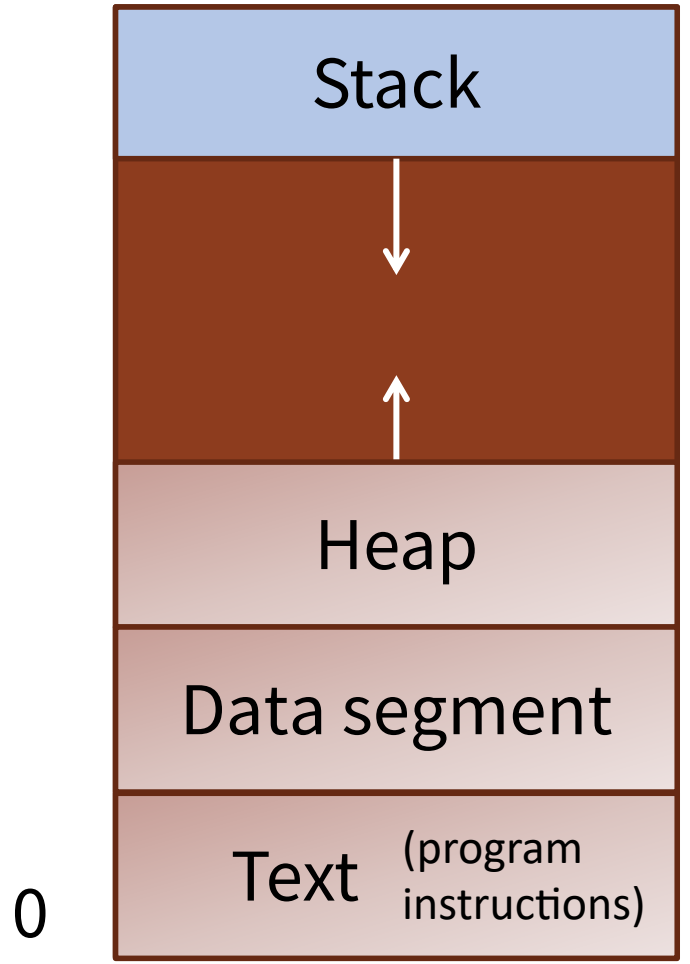
char []

- **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

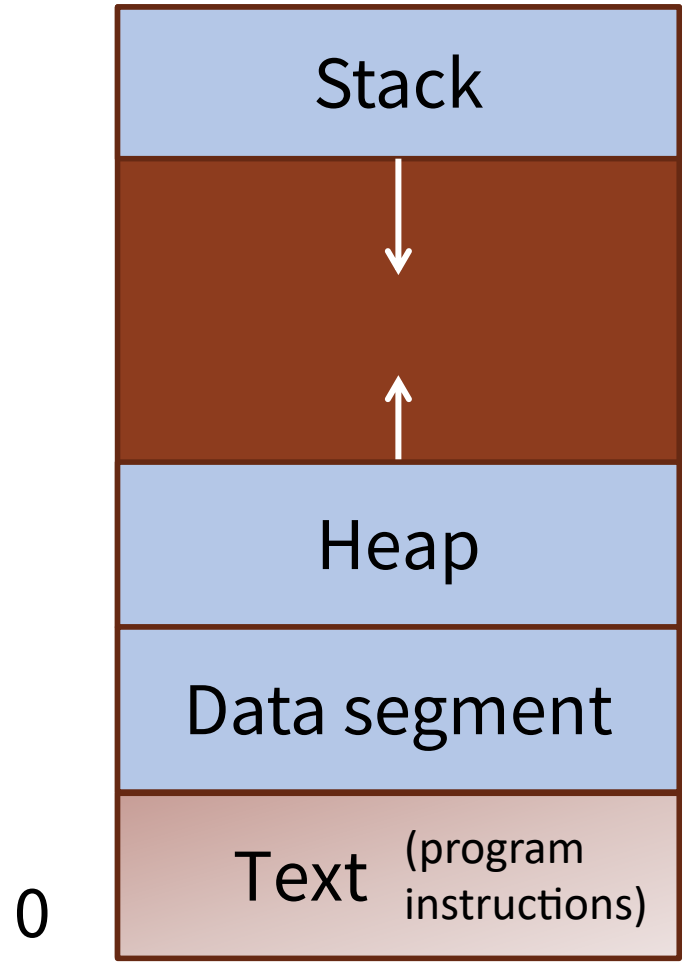
char *

- 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 <u>read-only</u> **string literals**.

```
// 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:

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

arr_ptr.c

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;
```

char* vs char[] exercises

Suppose we use a variable `str` as follows:

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

arr_ptr.c

For each of the following instantiations:

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

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 eulo3

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.

Super nitty gritty detail: `char *` vs `char[]`

Detail #2: Array initialization will <u>always</u> 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 `char`s' 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

Arrays of Strings

We can make an array of strings to group multiple strings together:

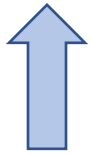
```
char *string_array[5];          // array of 5 char *s
```

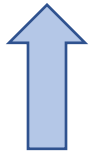
We can also use the following shorthand to initialize a string array:

```
char *string_array[] = {
    "hello world",              // strings
    "hi",                       // can have
    "have a nice day"           // different lengths
};
```

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):

```
void binky(char *x[]);
```

(suggestion): want to process an array of pointers

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

(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.

Skip spaces

Write a function **skip_spaces** that modifies a string pointer to skip past any leading spaces.

What should go in each of the blanks?

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

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

Skip spaces

Write a function **skip_spaces** that modifies a string pointer to skip past any leading spaces.

```
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`

Practice: Password Verification

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

```
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