CSE 29
Lecture 6 Summary

January 22, 2026

📣 Logistical Things

- Assignment 2 is due next Thursday (1/29), which includes:
  - PSet 2 on Prairie Learn
  - PA 2
  - Design Questions 2
  - We have gone over all the things you need to complete PA2
- Assignment 1 resubmission is also due next Thursday (1/29)
  - More details on resubmission policy: https://ucsd-cse29.github.io/wi26/#assignments

🧠 Review Questions

Answers in speaker notes!

For the string “Jéan” answer the following questions:

Q1: What is the UTF-8 encoding?

Q2: What is the UTF-32 encoding?

Q3: What is `strlen` of the UTF-8 encoding? The UTF-32?

J = 74 = 0x4A

é = 233 = 0xE9

a = 97 = 0x61

n = 110 = 0x6E

Answer 1: 0x4A 0xC3 0XA9 0X6E

strlen returns 5

Answer 2: 0x00 0x00 0x00 0x4A 0x00 0x00 0x00 0xE9 0x00 0x00 0x00 0x61 0x00 0x00 0x00 0x6E

strlen returns 0 for this version (strlen is NOT compatible with UTF-32 encoding!)

Another version of UTF-32 depending on byte ordering: 0x4A 0x00 0x00 0x00 0xE9 0x00 0x00 0x00 0x61 0x00 0x00 0x00 0x6E 0x00 0x00 0x00
https://en.wikipedia.org/wiki/UTF-32

strlen returns 1 for this version

Pointers

string.h

`int strlen(char *s)` class notation would be `int strlen(char s[])`

`void strcpy(char *dest, char *src)` Copies from src to dest

`void strcat(char *dest, char* src)` Appends src to end of dest

`char str[]` and `char *str` mean the same thing in function arguments

`char*`: an address in memory where we can access `char` data

`T*`: an address in memory where we can access T data

`T` could be `int, double, int16_t, char* etc`

Pointer Types

address = number representing a place in memory (storage in our computer)

For a pointer `T *p`:

- `p[index]` looks up the value offset by `index` from `p` in memory
- `p[index] = v` change the value offset by `index` from `p` to the value `v`

If `x` is a variable of type `T`

- `&x` => evaluates to a `T*` that is the **address** where `x` is stored

Questions

- Could you return pointers?
  - Yes! This is one of the things that differentiates `char[]` and `char*`.
- Are pointers also stored at an address?
  - Yes, pointers themselves also live somewhere in memory so they have addresses.

Look through this code, what observations do you have?

```c
#include <stdio.h>
#include <stdint.h>

void wheresmystuff(char* s) {
  int x = 12;
  uint8_t y = 53;
  char z = 9;
  double ns[] = { 4.0, 3.0, 9.0 };

  printf("x=%d, %lu bytes, starts at: %p\n", x, sizeof(x), &x);
  printf("y=%hhu, %lu bytes, starts at: %p\n", y, sizeof(y), &y);
  printf("z=%hhd, %lu bytes, starts at: %p\n", z, sizeof(z), &z);
  printf("ns=[%f,%f,%f], %lu bytes, starts at: %p\n",
         ns[0], ns[1], ns[2], sizeof(ns), &ns);

  printf("s=\"%s\"@%p, %lu bytes, starts at: %p\n", s, s,
      sizeof(s), &s);
}

int main() {
  char str[] = "14 char string";
  wheresmystuff(str);
  printf("\nstr takes up %lu bytes starting at: %p\n",
      sizeof(str), &str);
}
```

Output

```
$ # NOTE - Joe ran this on his Macbook to get this output
$ gcc wheresmystuff.c -o wheresmystuff
$ ./wheresmystuff
x=12, 4 bytes, starts at: 0x16f4c6e44
y=53, 1 bytes, starts at: 0x16f4c6e43
z=9, 1 bytes, starts at: 0x16f4c6e42
ns=[4.000000,3.000000,9.000000], 24 bytes, starts at:
    0x16f4c6e50
s="14 char string"@0x16f4c6e98, 8 bytes, starts at:
    0x16f4c6e48

str takes up 15 bytes starting at: 0x16f4c6e98
```

`%p` to print pointers

`%f` to print floating-point number (contrast from `%d`)

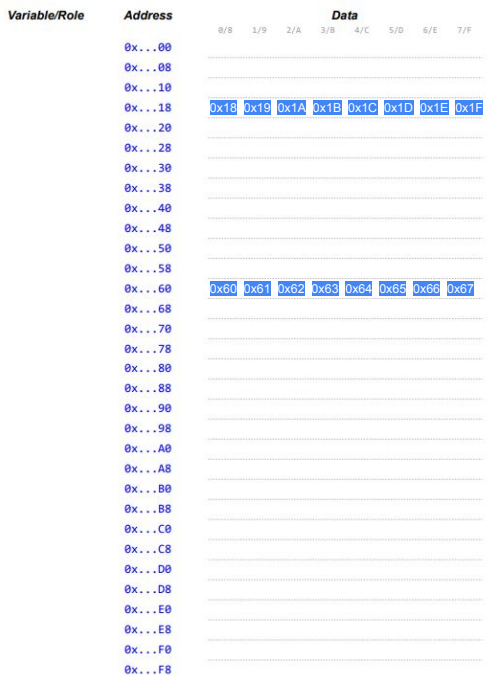
Observations

- In the print at line 16, it's interesting that `s` and `&s` are printing different things for the format specifier `%p`
  - `%p` prints out pointers, `&s` is the address of `s`
- `x`,`y`,`z` seem to be stored close together (based on the address)
  - Yeah, it's interesting how `x` is at 0x16f4c6e**44**, `y` is at 0x16f4c6e**43**, and z is at 0x16f4c6e**42**
- `sizeof` operator returns the size of the entire array and not the pointer for `ns`
  - There are three 8-byte doubles = 24 bytes
- `s` is a 15 character array, but its size is 8 bytes. Why is it not 15?
  - This is because in the context of the function, `s` is a pointer and not an array.

Memory Diagram

We will be using this memory diagram for the rest of the quarter to visualize what's going on inside the computer!

- Each row is 8 bytes of data
- Convenient for mapping out memory and variable storage!



Where are the variables stored?

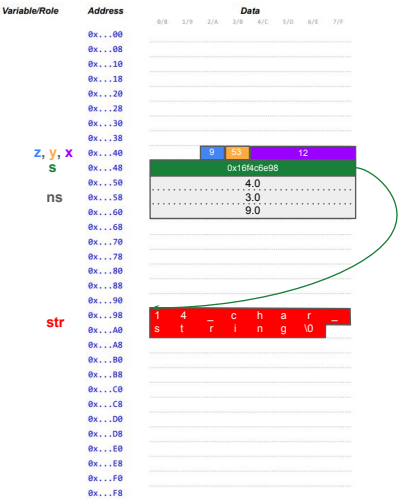
```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 void wheresmystuff(char* s) {
5   int x = 12;
6   uint8_t y = 53;
7   char z = 9;
8   double ns[] = { 4.0, 3.0, 9.0 };
9
10  printf("x=%d, %lu bytes, starts at: %p\n", x, sizeof(x), &x);
11  printf("y=%hhu, %lu bytes, starts at: %p\n", y, sizeof(y), &y);
12  printf("z=%hhd, %lu bytes, starts at: %p\n", z, sizeof(z), &z);
13  printf("ns=[%f,%f,%f], %lu bytes, starts at: %p\n",
14         ns[0], ns[1], ns[2], sizeof(ns), &ns);
15
16  printf("s=\"%s\"@%p, %lu bytes, starts at: %p\n", s, s,
        sizeof(s), &s);
17 }
18
19 int main() {
20   char str[] = "14 char string";
21   wheresmystuff(str);
22   printf("\nstr takes up %lu bytes starting at: %p\n",
        sizeof(str), &str);
23 }
```

```
$ # NOTE - Joe ran this on his Macbook to get this output
$ gcc wheresmystuff.c -o wheresmystuff
$ ./wheresmystuff
x=12, 4 bytes, starts at: 0x16f4c6e44
y=53, 1 bytes, starts at: 0x16f4c6e43
z=9, 1 bytes, starts at: 0x16f4c6e42
ns=[4.000000,3.000000,9.000000], 24 bytes, starts at:
    0x16f4c6e50
s="14 char string"@0x16f4c6e98, 8 bytes, starts at:
    0x16f4c6e48

str takes up 15 bytes starting at: 0x16f4c6e98
```

Where are the variables stored? (Answer)

```c
#include <stdio.h>
#include <stdint.h>

void wheresmystuff(char* s) {
  int x = 12;
  uint8_t y = 53;
  char z = 9;
  double ns[] = { 4.0, 3.0, 9.0 };

  printf("x=%d, %lu bytes, starts at: %p\n", x, sizeof(x), &x);
  printf("y=%hhu, %lu bytes, starts at: %p\n", y, sizeof(y), &y);
  printf("z=%hhd, %lu bytes, starts at: %p\n", z, sizeof(z), &z);
  printf("ns=[%f,%f,%f], %lu bytes, starts at: %p\n",
         ns[0], ns[1], ns[2], sizeof(ns), &ns);

  printf("s=\"%s\"@%p, %lu bytes, starts at: %p\n", s, s,
      sizeof(s), &s);
}

int main() {
  char str[] = "14 char string";
  wheresmystuff(str);
  printf("\nstr takes up %lu bytes starting at: %p\n",
      sizeof(str), &str);
}
```

```
$ # NOTE - Joe ran this on his Macbook to get this output
$ gcc wheresmystuff.c -o wheresmystuff
$ ./wheresmystuff
x=12, 4 bytes, starts at: 0x16f4c6e44
y=53, 1 bytes, starts at: 0x16f4c6e43
z=9, 1 bytes, starts at: 0x16f4c6e42
ns=[4.000000,3.000000,9.000000], 24 bytes, starts at:
    0x16f4c6e50
s="14 char string"@0x16f4c6e98, 8 bytes, starts at:
    0x16f4c6e48

str takes up 15 bytes starting at: 0x16f4c6e98
```



Questions

- How does the program decide where to put variables?
  - There is an algorithm in gcc to decide this. It would be different on different versions of gcc and computers
- How does sizeof know the size?
  - `sizeof` is a compile-time operator, it means that the compiler calculates the value while it is building your program, not while the program is actually running. As it compiles, a size is determined for every variable

Hashing

Examples of generated hashes

```
$ git commit -m “almost done”
[main 211935b] almost done
1 file changed…
$ git log
Commit 211935b0cf003 … (40 hex char = 20 bytes)
$ ssh jpolitz@ieng6
ED25519 key fingerprint SHA256: 8avDdt0
```

You have seen this in lab or doing the PA!

How are these numbers generated?

`SHA256(char *input, int len char* hash)`

More info in speaker notes 💡

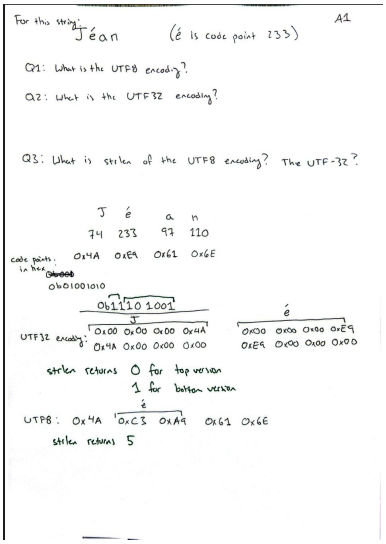
- `input` is not a C-string and doesn't need to be null terminated
- `len` is the number of chars to process with the hashing function
- `hash` is an output parameter that is written over

Calculates a fixed-size (32 byte) "hash" of that input data.

- deterministic (NOT random)
- "one-way": hard to guess the input from output
- unpredictably distributed: similar inputs produce very different outputs

char *input is JUST a byte array, not a C string. Not every char* you see will be a C string.

Joe's Notes (11am)



A1

For this string: Jéan (é is code point 233)

Q1: What is the UTF8 encoding?

Q2: What is the UTF32 encoding?

Q3: What is strlen of the UTF8 encoding? The UTF-32?

J é a n
74 233 97 110

code points in hex: 0x4A 0xE9 0x61 0x6E

0b01001010

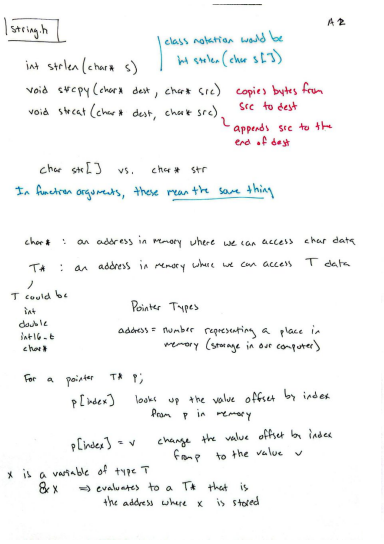
0b11101001

UTF32 encoding: 0x00 0x00 0x00 0x4A | 0x00 0x00 0x00 0xE9
0x4A 0x00 0x00 0x00 | 0xE9 0x00 0x00 0x00

strlen returns 0 for top version
1 for bottom version

UTF8: 0x4A 0xC3 0xA9 0x61 0x6E

strlen returns 5



A2

String.h

int strlen(char* s) | class notation would be int strlen(char s[])

void strcpy(char* dest, char* src) copies bytes from src to dest

void strcat(char* dest, char* src) appends src to the end of dest

char str[] vs. char* str

In function arguments, these mean the same thing

char* : an address in memory where we can access char data

T* : an address in memory where we can access T data

T could be int, double, int16_t, char*

Pointer Types

address = number representing a place in memory (storage in our computer)

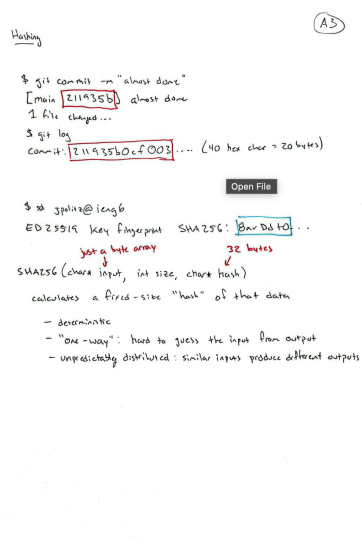
For a pointer T* p;

p[index] looks up the value offset by index from p in memory

p[index] = v change the value offset by index from p to the value v

x is a variable of type T

&x ⇒ evaluates to a T* that is the address where x is stored



A3

Hashing

$ git commit -m "almost done"
[main 2119356b] almost done
1 file changed...
$ git log
commit: 2119356bcf003... (40 hex char = 20 bytes)

Open File

$ ssh jpaliz@ieng6
ED25519 key fingerprint SHA256: Bar Dd tO...

just a byte array — 32 bytes

SHA256(char* input, int size, char* hash)

calculates a fixed-size "hash" of that data

- deterministic
- "one-way": hard to guess the input from output
- unpredictably distributed: similar inputs produce different outputs

Joe's Notes (12:30pm)