CSE 29
Lecture 12 Summary

February 12, 2026

📣 Logistical Things

- Nothing today!

Review Questions

Answers in next slide!

```
typedef struct Len32String {
  char contents[32];
} Len32String;

Len32String return_struct() {
  Len32String str_struct = { "some string content" };
  return str_struct;
}

char* return_array() {
  char str[] = "some string content";
  return str;
}

void example2() {
  char* from_array1 = return_array();
  printf("from_array1:\t%p\n", from_array1);
  char* from_array2 = return_array();
  printf("from_array2:\t%p\n", from_array2);

  Len32String from_struct1 = return_struct();
  printf("from_struct1:\t%p\n", from_struct1.contents);
  Len32String from_struct2 = return_struct();
  printf("from_struct2:\t%p\n", from_struct2.contents);
}
```

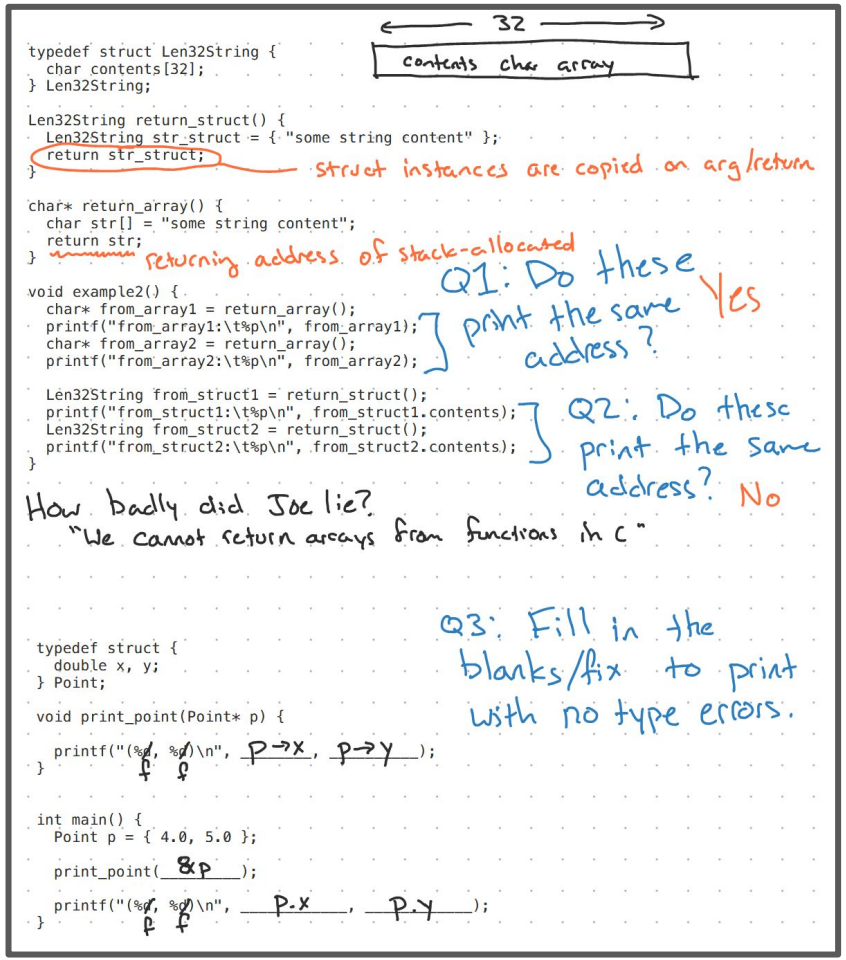
Q1: Could these print the same address?

Q2: Could these print the same address?

```
typedef struct {
  double x, y;
} Point;

void print_point(Point* p) {

  printf("(%d, %d)\n", _______, _________);
}


int main() {
  Point p = { 4.0, 5.0 };

  print_point(________);

  printf("(%d, %d)\n", ___________, ___________);
}
```

Q3: Fill in blanks/ fix to print w/no type errors

Review Question Answers



We just returned an array from a function because we just wrapped in in a struct (did Joe lie to us? 😱)

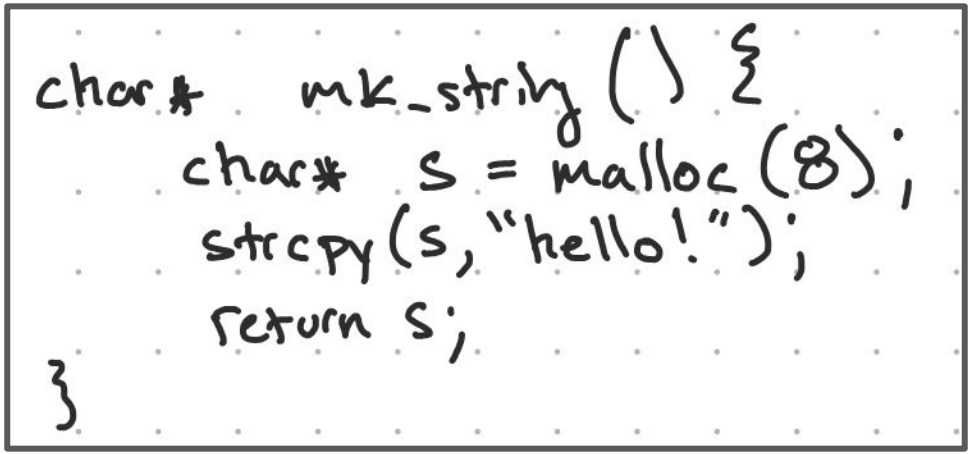
- However, we would not be able to return variable-sized arrays
- The struct is already initialized to have an array of a specific size

malloc

`void* malloc(size_t size)`

- Function Declaration
  - `void*` is the type of pointers with unknown underlying type. It is assignable to any pointer type
  - `size_t` is just an `int64_t`
- `malloc` sets aside `size` bytes on **the heap** and returns the address of the start of that space
- If there's no more space, malloc returns `NULL`
  - That's why we don't use malloc if it's life-or-death related software and your program can not crash
- What's special about the heap is that what is written on the heap will persist and not get overwritten (like stack-allocated things)

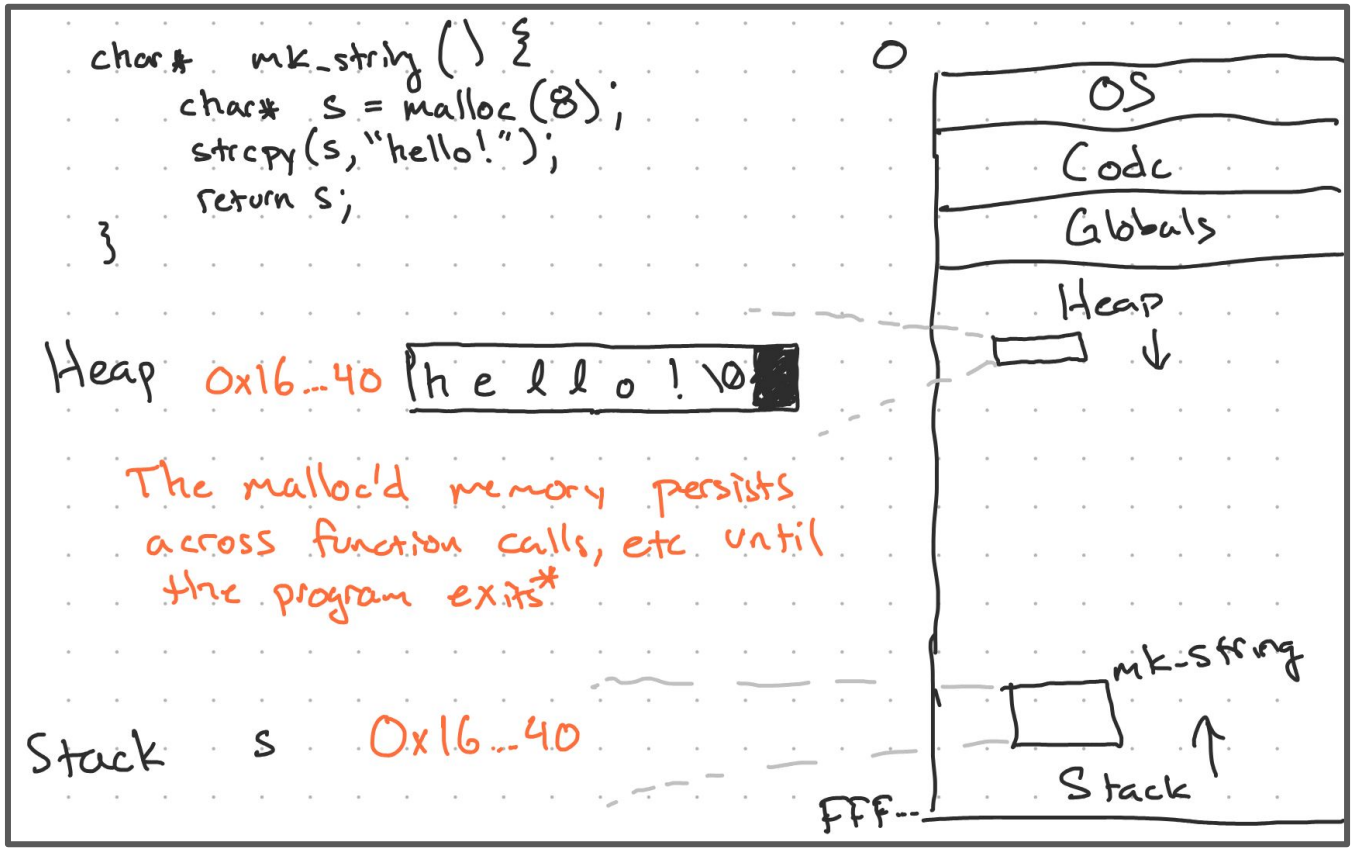
Lets try using `malloc()`!



(Typically, we don't put an actual number as the argument. We usually calculate something and then use that as an argument.)

- We got space (8 bytes) on **the heap** from `malloc`
- Now we can put things in this space!

`mk_string` in memory



In the heap, not the stack (like before!)

- the heap grows down
- the stack grows up

Lets try another function: `strdup`

`strdup` - takes in a string `s` (could be stack/global/heap allocated and return a heap allocated copy

```
char* strdup(char* s) {  //return a heap-allocated
                            copy of s




}
```

`strdup` Solution

```
char* strdup(char* s) {  // return a heap-allocated copy of s
    char* new_s = malloc(strlen(s) + 1);
    strcpy(new_s, s);
    return new_s;
}
```

Note: the argument to `malloc` is `strlen(s) + 1` to account for the NULL terminator

Questions

- Is the only difference between the heap and stack that you decide what to put where?
  - The main difference is that heap data will persist and not be overwritten ever
    - The heap memory can still get written over if the program ends or we decide that we do want to write over that location in memory
  - We had a lot of issues when working with arrays on the stack, and this is a solution!

Implementing Python Strings (with malloc)

Strings in C vs Strings in Python/Java

- C-strings suck
  - We need to keep track of how long they are
  - There's a null terminator we need to be aware of
  - We need to use strtok
- Python and Java strings have great features
  - We can concatenate with +
  - We can split it up into an array without a loop
  - We can find the length
  - They have great **abstraction**
- The abstractions are built in C! (Python and Java use structs for their strings!)

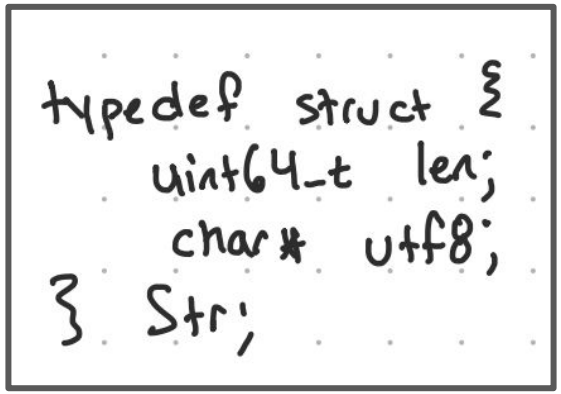
This is the struct for Python strings!

https://github.com/python/cpython/blob/main/Include/cpython/unicodeobject.h#L166

```
typedef struct {
    PyASCIIObject _base;
    Py_ssize_t utf8_length;     /* Number of bytes in utf8, excluding the
                                 * terminating \0. */
    char *utf8;                 /* UTF-8 representation (null-terminated) */
} PyCompactUnicodeObject;
```

this field usually is an address of a heap-allocated string!

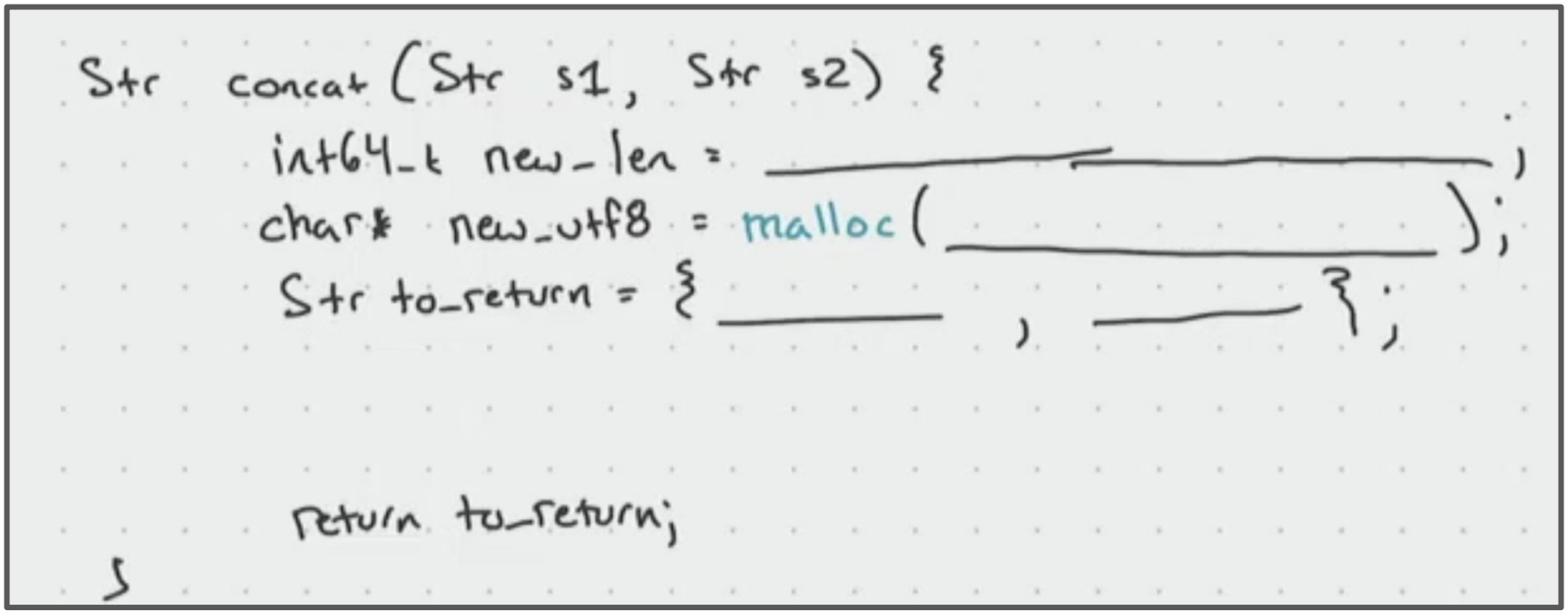
Lets use this for our class



We are now the programmers that created Python! Lets try implementing a feature for strings in Python..

Lets create a function for concatenation! (`concat`)

This is like if we did `s1 + s2` in Python
Fill in the blanks, add any line you think are needed!



`concat` Solution

We need the null terminator

```
Str concat(Str s1, Str s2) {
    int64_t new_len = s1.len + s2.len;
    char* new_utf8 = malloc(new_len + 1);
    Str to_return = { new_len, new_utf8 };
    strcpy(new_utf8, s1.utf8);
    strcat(new_utf8, s2.utf8);
    // strcpy(&new_utf8[s1.len], s2.utf8);
    return to_return;
}
```

Instead of just concatenating to s1.utf8, to concatenate to another string

- because we don't know if s1.utf8 is long enough to hold s2.utf8 as well
- s1 and s2 shouldn't be modified in here (strings are immutable)

Also valid

Questions

- Couldn’t we just call `strcat` on `new_utf8` twice?
  - new_utf8 doesn’t start out as “empty” (everything zeroed out)
  - The space that is given by malloc holds junk values from previous programs
- How does `strcat` work?
  - It loops through the first string until it finds the null terminator, replaces the null terminator with the first character of the second string, and goes on until the null terminator of the second string

Let's write a function that creates a new `Str`

When we write a string in python, something like this function is called

```
Str new_str(char* s) {
    char* duped = strdup(s);
    Str str = {strlen(s), duped};
    return str;
}
```

We wrote this earlier!

free

Lets try writing this small Python program in C

```
def concat_all(strs):
    s = ""
    for str in strs:
        s = s + str
    return s
```

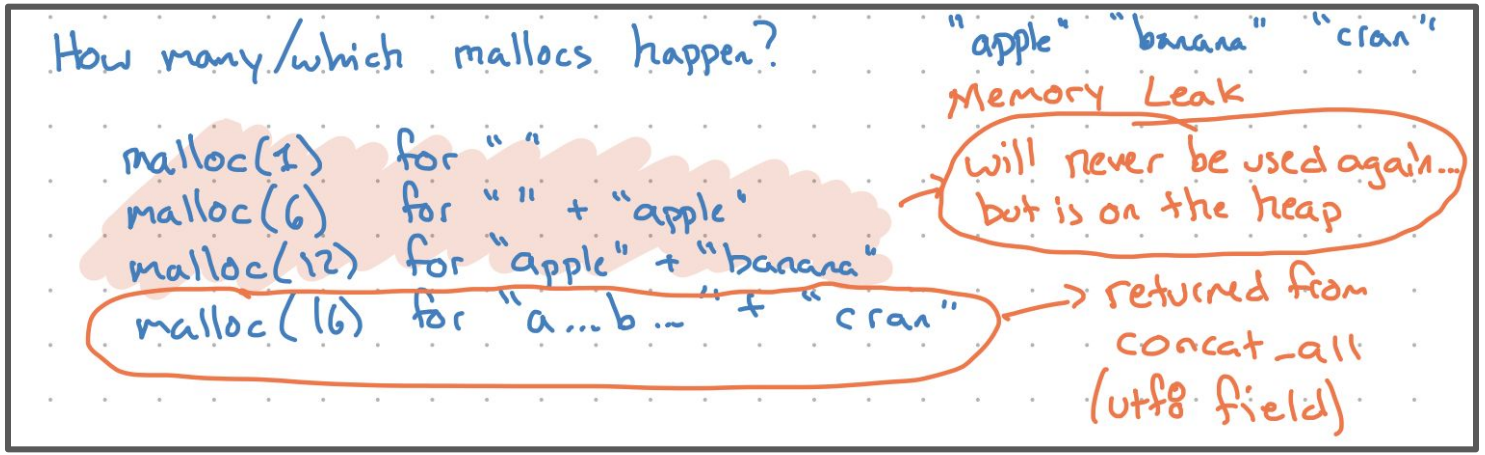
```
Str concat_all(Str strs[], int count) {
    Str s = new_str("");
    for(int i=0; i<count; i+=1) {
        char* old_utf8 = s.utf8;
        s = concat(s, strs[i]);
        free(old_utf8);
    }
    return s;
}
```

Kind of how `join` is implemented in Python

What is this?! (next slides)

Here's how the iterations look

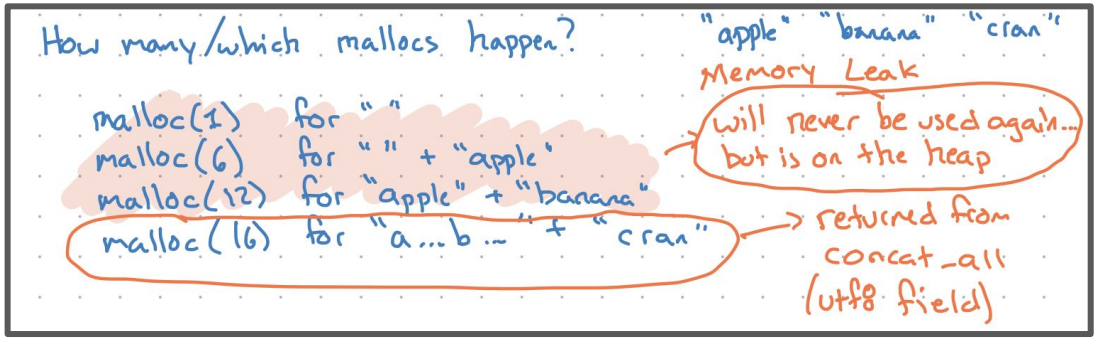
```
Str concat_all(Str strs[], int count) {
    Str s = new_str("");
    for(int i=0; i<count; i+=1) {
        char* old_utf8 = s.utf8;
        s = concat(s, strs[i]);
        free(old_utf8);
    }
    return s;
}
```



Memory Leak

We have things in the heap that we no longer need!

- We need to get rid of them or else we will get a memory leak
- Computers have a finite amount of memory, so we need to make sure to let go of junk
- We need to free up memory and tell our computer that it can use that memory again



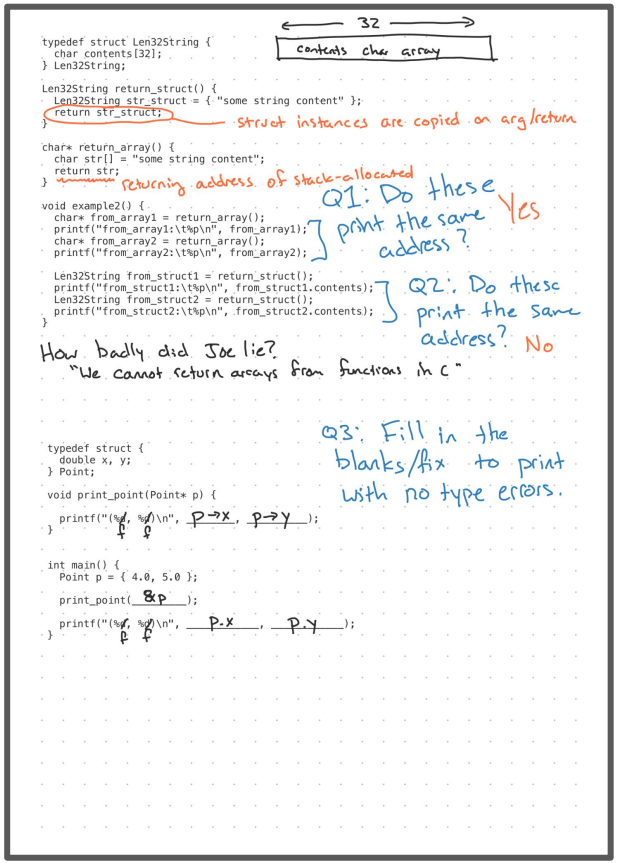
`void free(void *ptr)`

- Takes a pointer returned by `malloc` and tells `malloc` it can reuse that space

```
Str concat_all(Str strs[], int count) {
    Str s = new_str("");
    for(int i=0; i<count; i+=1) {
        char* old_utf8 = s.utf8;
        s = concat(s, strs[i]);
        free(old_utf8);
    }
    return s;
}
```

- Lot of security vulnerabilities with `free` (ask Aaron Schulman)

Joe's Notes (11am)



```
typedef struct Len32String {
  char contents[32];
} Len32String;

Len32String return_struct() {
  Len32String str_struct = { "some string content" };
  return str_struct;
}

char* return_array() {
  char str[] = "some string content";
  return str;
}

void example2() {
  char* from_array1 = return_array();
  printf("from_array1:\t%p\n", from_array1);
  char* from_array2 = return_array();
  printf("from_array2:\t%p\n", from_array2);

  Len32String from_struct1 = return_struct();
  printf("from_struct1:\t%p\n", from_struct1.contents);
  Len32String from_struct2 = return_struct();
  printf("from_struct2:\t%p\n", from_struct2.contents);
}
```

32 — contents char array

struct instances are copied on arg/return

returning address of stack-allocated

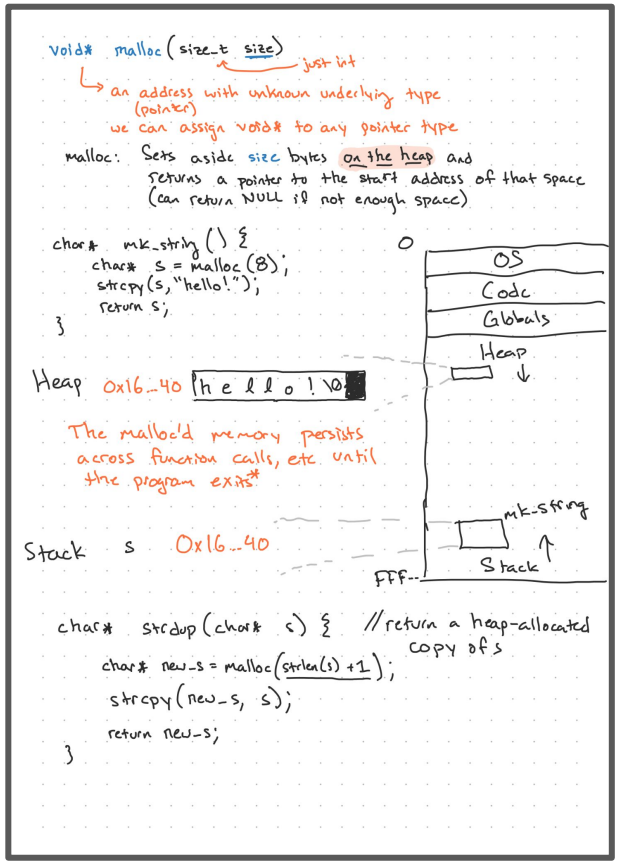
Q1: Do these print the same address? Yes

Q2: Do these print the same address? No

How badly did Joe lie?
"We cannot return arrays from functions in C"

```
typedef struct {
  double x, y;
} Point;

void print_point(Point* p) {
  printf("(%f, %f)\n", p->x, p->y);
}

int main() {
  Point p = { 4.0, 5.0 };

  print_point(&p);

  printf("(%f, %f)\n", p.x, p.y);
}
```

Q3: Fill in the blanks/fix to print with no type errors.



```
void* malloc(size_t size)
```

size_t: just int

↳ an address with unknown underlying type (pointer)
we can assign void* to any pointer type

malloc: Sets aside size bytes on the heap and returns a pointer to the start address of that space (can return NULL if not enough space)

```
char* mk_string() {
  char* s = malloc(8);
  strcpy(s, "hello!");
  return s;
}
```

Heap 0x16...40 h e l l o ! \0

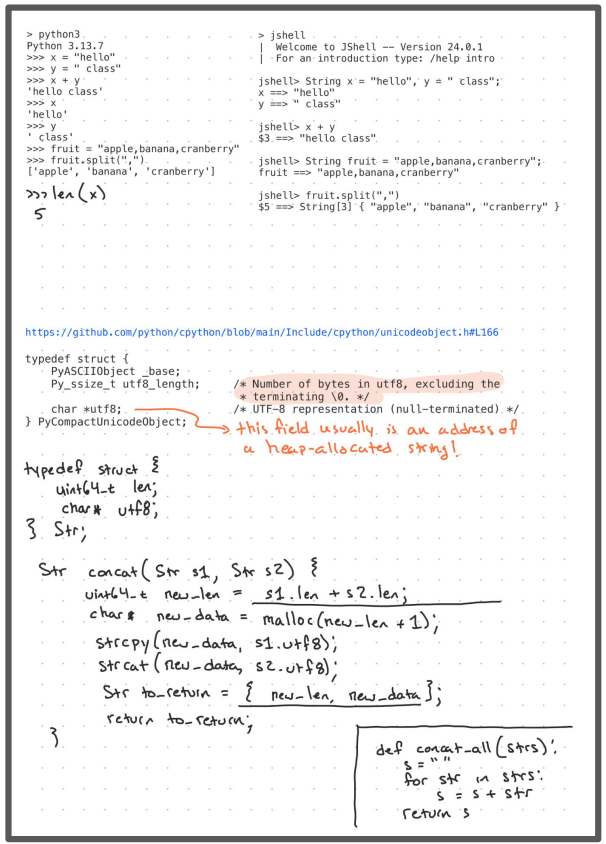
The malloc'd memory persists across function calls, etc until the program exits*

Stack s 0x16...40

0 — OS / Code / Globals / Heap / mk_string / Stack / FFF...

```
char* strdup(char* s) {  // return a heap-allocated copy of s
  char* new_s = malloc(strlen(s) + 1);
  strcpy(new_s, s);
  return new_s;
}
```

Joe's Notes (11am)



```
> python3
Python 3.13.7
>>> x = "hello"
>>> y = " class"
>>> x + y
'hello class'
>>> x
'hello'
>>> y
' class'
>>> fruit = "apple,banana,cranberry"
>>> fruit.split(",")
['apple', 'banana', 'cranberry']
>>> len(x)
5
```

```
> jshell
|  Welcome to JShell -- Version 24.0.1
|  For an introduction type: /help intro

jshell> String x = "hello", y = " class";
x ==> "hello"
y ==> " class"

jshell> x + y
$3 ==> "hello class"

jshell> String fruit = "apple,banana,cranberry";
fruit ==> "apple,banana,cranberry"

jshell> fruit.split(",")
$5 ==> String[3] { "apple", "banana", "cranberry" }
```

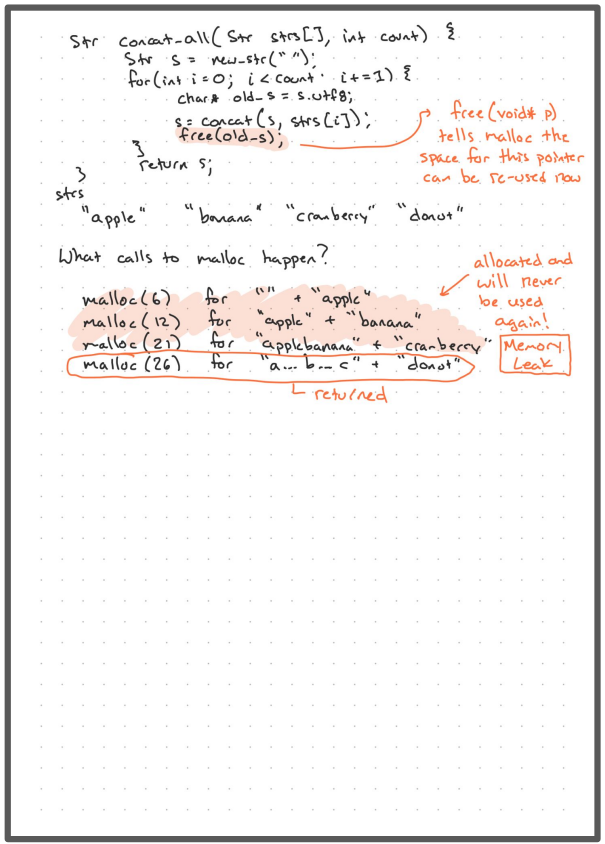
https://github.com/python/cpython/blob/main/Include/cpython/unicodeobject.h#L166

```
typedef struct {
    PyASCIIObject _base;
    Py_ssize_t utf8_length;     /* Number of bytes in utf8, excluding the
                                 * terminating \0. */
    char *utf8;                 /* UTF-8 representation (null-terminated) */
} PyCompactUnicodeObject;
```

this field usually is an address of a heap-allocated string!

```
typedef struct {
    uint64_t len;
    char* utf8;
} Str;

Str concat(Str s1, Str s2) {
    uint64_t new_len = s1.len + s2.len;
    char* new_data = malloc(new_len + 1);
    strcpy(new_data, s1.utf8);
    strcat(new_data, s2.utf8);
    Str to_return = { new_len, new_data };
    return to_return;
}
```

```
def concat_all(strs):
    s = ""
    for str in strs:
        s = s + str
    return s
```



```
Str concat_all(Str strs[], int count) {
    Str s = new_str("");
    for(int i=0; i<count; i+=1) {
        char* old_s = s.utf8;
        s = concat(s, strs[i]);
        free(old_s);
    }
    return s;
}
```

free(void* p) tells malloc the space for this pointer can be re-used now

strs: "apple" "banana" "cranberry" "donut"

What calls to malloc happen?

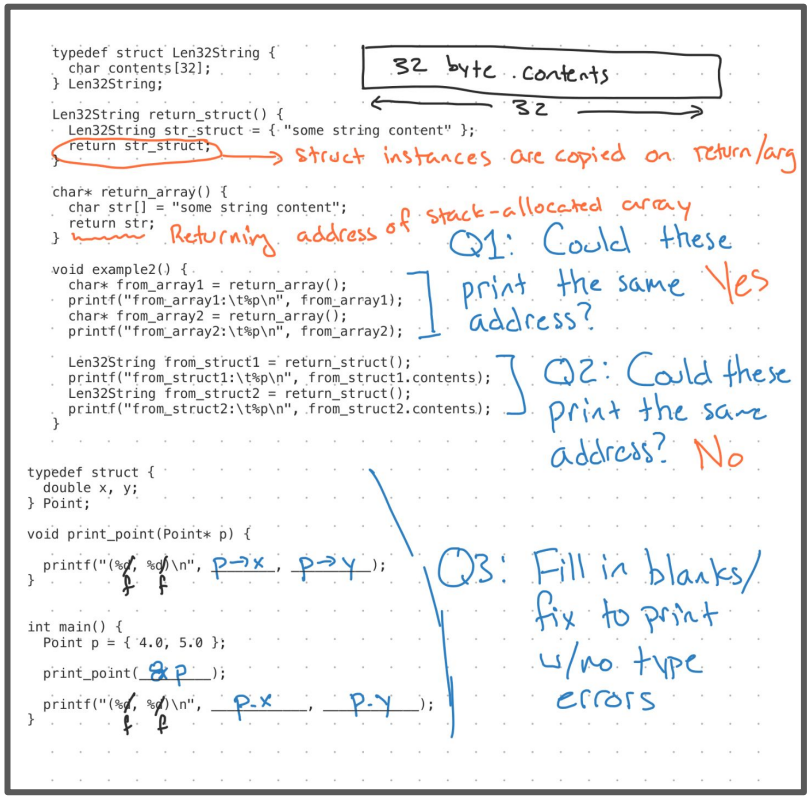
```
malloc(6)   for "" + "apple"
malloc(12)  for "apple" + "banana"
malloc(21)  for "applebanana" + "cranberry"
malloc(26)  for "a...b...c" + "donut"
```

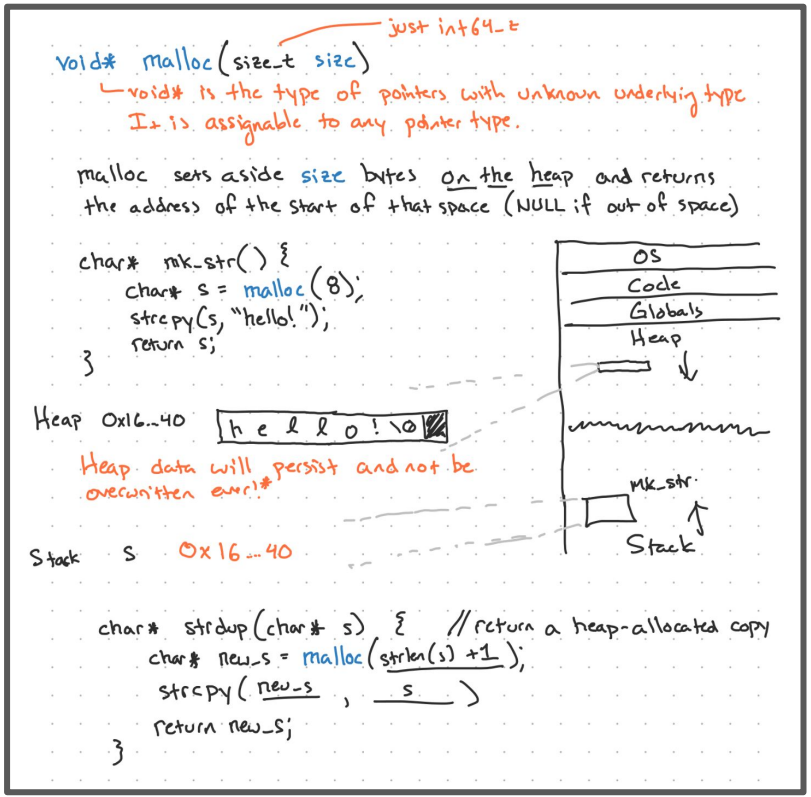
allocated and will never be used again!

Memory Leak

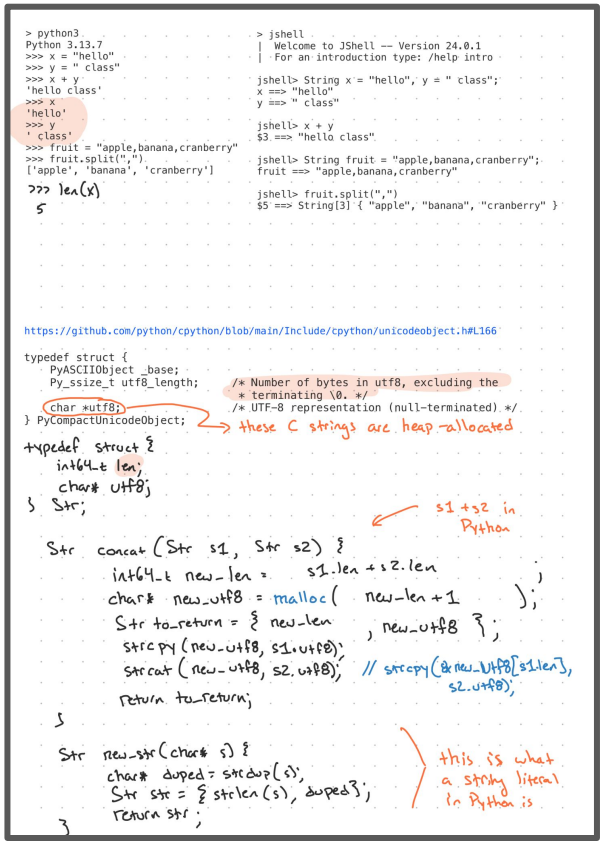
returned

Joe's Notes (12:30pm)





Joe's Notes (12:30pm)



```
> python3
Python 3.13.7
>>> x = "hello"
>>> y = " class"
>>> x + y
'hello class'
>>> x
'hello'
>>> y
' class'
>>> fruit = "apple,banana,cranberry"
>>> fruit.split(",")
['apple', 'banana', 'cranberry']
>>> len(x)
5
```

```
> jshell
|  Welcome to JShell -- Version 24.0.1
|  For an introduction type: /help intro

jshell> String x = "hello", y = " class";
x ==> "hello"
y ==> " class"

jshell> x + y
$3 ==> "hello class"

jshell> String fruit = "apple,banana,cranberry";
fruit ==> "apple,banana,cranberry"

jshell> fruit.split(",")
$5 ==> String[3] { "apple", "banana", "cranberry" }
```

https://github.com/python/cpython/blob/main/Include/cpython/unicodeobject.h#L166

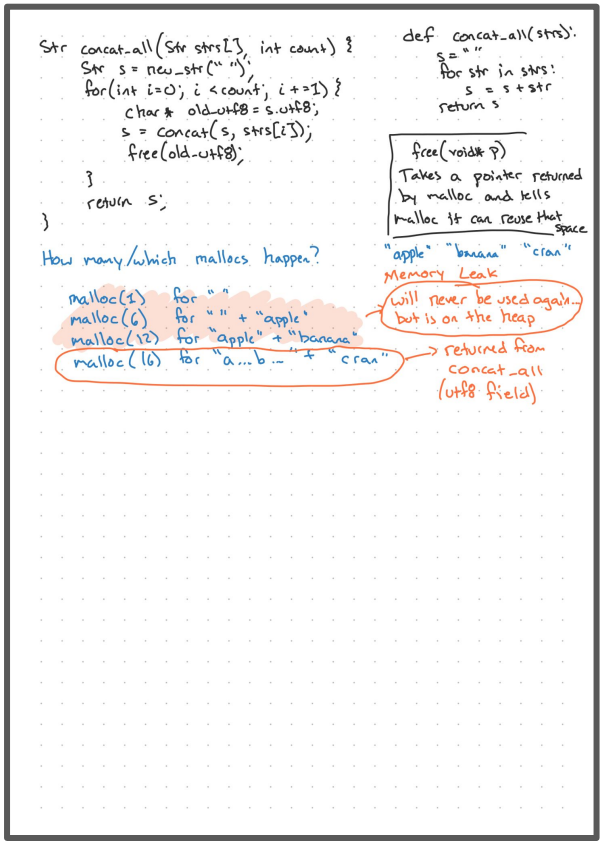
```
typedef struct {
    PyASCIIObject _base;
    Py_ssize_t utf8_length;     /* Number of bytes in utf8, excluding the
                                 * terminating \0. */
    char *utf8;                 /* UTF-8 representation (null-terminated) */
} PyCompactUnicodeObject;
```

these C strings are heap-allocated

```
typedef struct {
    int64_t len;
    char* utf8;
} Str;

Str concat(Str s1, Str s2) {
    int64_t new_len = s1.len + s2.len;
    char* new_utf8 = malloc(new_len + 1);
    Str to_return = {new_len, new_utf8};
    strcpy(new_utf8, s1.utf8);
    strcat(new_utf8, s2.utf8);  // strcpy(&new_utf8[s1.len], s2.utf8);
    return to_return;
}

Str new_str(char* s) {
    char* duped = strdup(s);
    Str str = {strlen(s), duped};
    return str;
}
```

s1 + s2 in Python

this is what a string literal in Python is



```
Str concat_all(Str strs[], int count) {
    Str s = new_str("");
    for(int i=0; i<count; i++) {
        char* old_utf8 = s.utf8;
        s = concat(s, strs[i]);
        free(old_utf8);
    }
    return s;
}
```

```
def concat_all(strs):
    s = ""
    for str in strs:
        s = s + str
    return s
```

free(void* p)
Takes a pointer returned by malloc and tells malloc it can reuse that space

How many/which mallocs happen?

"apple" "banana" "cran"

malloc(1) for ""
malloc(6) for "" + "apple"
malloc(12) for "apple" + "banana"
malloc(16) for "a...b..." + "cran"

Memory Leak
Will never be used again... but is on the heap

returned from concat_all (utf8 field)