0 vs NULL // Twitter @raysato

COMP201

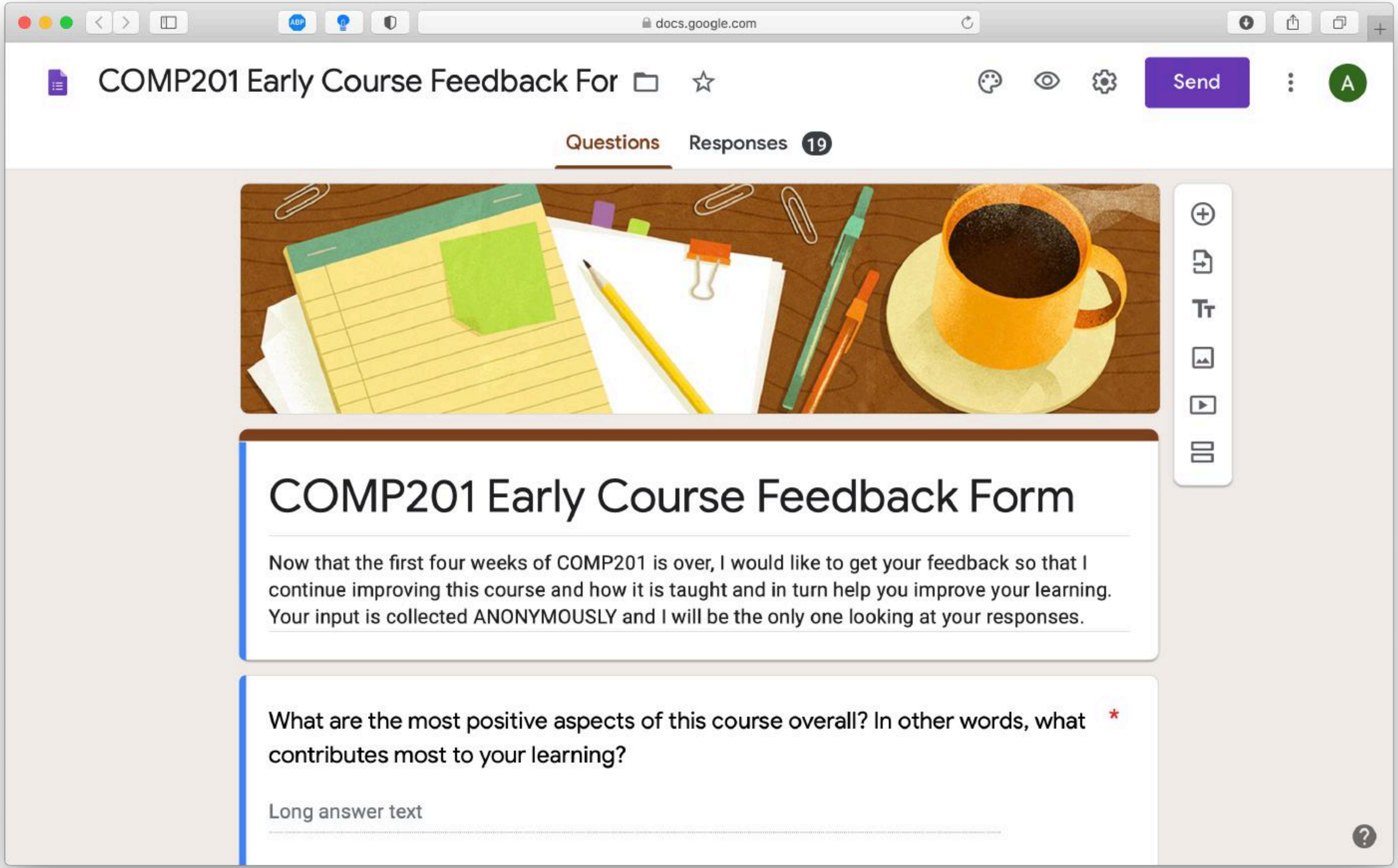
Computer Systems & Programming

Lecture #12 – Other heap allocations, `realloc`

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2020

If you are badly affected by the earthquake at Izmir, please feel free to contact me or your TA. We can give you a make-up for Lab 3 or Quiz 5.



https://forms.gle/5d8LDWfH84pC33Fg7

Recap

- Pointer Arithmetic
- The Stack
- The Heap and Dynamic Memory

Plan for Today

- Other heap allocations
- `realloc`

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class

Lecture Plan

- Other heap allocations
- `realloc`

Recap: `malloc`

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

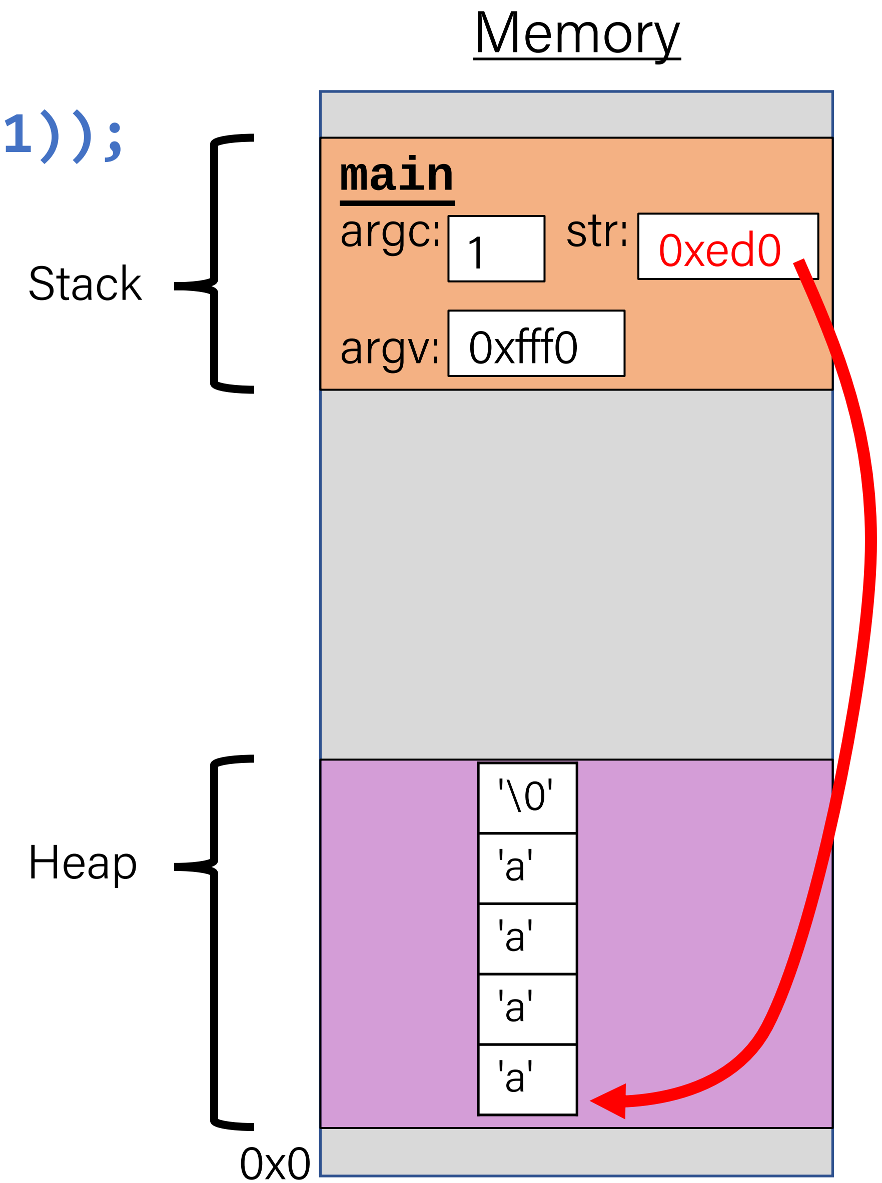
To allocate memory on the heap, use the **`malloc`** function (“memory allocate”) and specify the number of bytes you’d like.

- This function returns a pointer to *the **starting address** of the new memory*. It doesn’t know or care whether it will be used as an array, a single block of memory, etc.
- **`void *`**means a pointer to generic memory. You can set another pointer equal to it without any casting.
- The memory is *not* cleared out before being allocated to you!
- If `malloc` returns `NULL`, then there wasn’t enough memory for this request.

Recap: malloc

```
char *create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char) * (num + 1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('a', 4);
    printf("%s", str);  // want "aaaa"
    return 0;
}
```



Other heap allocations: `calloc`

```
void *calloc(size_t nmemb, size_t size);
```

`calloc` is like `malloc` that **zeros out** the memory for you—thanks, `calloc`!

- You might notice its interface is also a little different—it takes two parameters, which are multiplied to calculate the number of bytes (`nmemb * size`).

```
// allocate and zero 20 ints
int *scores = calloc(20, sizeof(int));

// alternate (but slower)
int *scores = malloc(20 * sizeof(int));
for (int i = 0; i < 20; i++) scores[i] = 0;
```

- calloc is more expensive than malloc because it zeros out memory. Use only when necessary!

Other heap allocations: `strdup`

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

`strdup` is a convenience function that returns a **null-terminated**, heap-allocated string with the provided text, instead of you having to **`malloc`** and copy in the string yourself.

```c
char *str = strdup("Hello, world!");  // on heap
str[0] = 'h';
```

Implementing strdup

How can we implement **strdup** using functions we've already seen?

```
char *myStrdup(char *str) {
    char *heapStr = malloc(strlen(str) + 1);
    assert(heapStr != NULL);
    strcpy(heapStr, str);
    return heapStr;
}
```

Cleaning Up with `free`

```
void free(void *ptr);
```

- If we allocated memory on the heap and no longer need it, it is our responsibility to **delete** it.
- To do this, use the **`free`** command and pass in the *starting address on the heap for the memory you no longer need.*
- Example:

```
char *bytes = malloc(4);
…
free(bytes);
```

`free` details

Even if you have multiple pointers to the same block of memory, each memory block should only be freed **once**.

```
char *bytes = malloc(4);
char *ptr = bytes;
…
free(bytes);      ✅
…
free(ptr);        ❌ Memory at this address was already freed!
```

You must free the address you received in the previous allocation call; you cannot free just part of a previous allocation.

```
char *bytes = malloc(4);
char *ptr   = malloc(10);
…
free(bytes);      ✅
…
free(ptr + 1);    ❌
```

Cleaning Up

You may need to free memory allocated by other functions if that function expects the caller to handle memory cleanup.

```
char *str = strdup("Hello!");
…
free(str);    // our responsibility to free!
```

Memory Leaks

- A memory leak is when you allocate memory on the heap, but do not free it.
- Your program should be responsible for cleaning up any memory it allocates but no longer needs.
- If you never free any memory and allocate an extremely large amount, you may run out of memory in the heap!

However, memory leaks rarely (if ever) cause crashes.

- We recommend not to worry about freeing memory until your program is written. Then, go back and free memory as appropriate.
- Valgrind is a very helpful tool for finding memory leaks!

free Practice

Freeing Memory

Where should we free memory below so that all memory is freed properly?

```
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
    int *num = malloc(sizeof(int));
    assert(num != NULL);
    *num = i;
    printf("%s %d\n", ptr, *num);
}
printf("%s\n", str);
```

Freeing Memory

Where should we free memory below so that all memory is freed properly?

```c
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
    int *num = malloc(sizeof(int));
    assert(num != NULL);
    *num = i;
    printf("%s %d\n", ptr, *num);
}
printf("%s\n", str);
```

NEW Head for **www.slido.com** and enter your answers using the event code **#73165**

Freeing Memory

Where should we free memory below so that all memory is freed properly?

```
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
     int *num = malloc(sizeof(int));
     assert(num != NULL);
     *num = i;
     printf("%s %d\n", ptr, *num);
     free(num);
}
printf("%s\n", str);
free(str);
```

Demo: Pig Latin

`pig_latin.c`

realloc

```c
void *realloc(void *ptr, size_t size);
```

- The **realloc** function takes an existing allocation pointer and enlarges to a new requested size. It returns the new pointer.
- If there is enough space after the existing memory block on the heap for the new size, **realloc** simply adds that space to the allocation.
- If there is not enough space, **realloc** *moves the memory to a larger location*, frees the old memory for you, and *returns a pointer to the new location*.

realloc

```
char *str = strdup("Hello");
assert(str != NULL);
…

// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);

strcat(str, addition);
printf("%s", str);
free(str);
```

realloc

- realloc only accepts pointers that were previously returned my malloc/etc.
- Make sure to not pass pointers to the middle of heap-allocated memory.
- Make sure to not pass pointers to stack memory.

Cleaning Up with `free` and `realloc`

You only need to free the new memory coming out of `realloc` —the previous (smaller) one was already reclaimed by `realloc`.

```
char *str = strdup("Hello");
assert(str != NULL);
…
// want to make str longer to hold "Hello world!"
char *addition = " world!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
assert(str != NULL);
strcat(str, addition);
printf("%s", str);
free(str);
```

Heap allocator analogy: A hotel

Request memory by size (`malloc`)

- Receive room key to first of connecting rooms

Need more room? (`realloc`)

- Extend into connecting room if available
- If not, trade for new digs, employee moves your stuff for you

Check out when done (`free`)

- You remember your room number though

Errors! What happens if you...

- Forget to check out?
- Bust through connecting door to neighbor? What if the room is in use? Yikes...
- Return to room after checkout?

Demo: Pig Latin Part 2

`pig_latin.c`

Heap allocation interface: A summary

```c
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Compare and contrast the heap memory functions we've learned about.

Heap allocation interface: A summary

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
char *strdup(char *s);
void free(void *ptr);
```

Heap **memory allocation** guarantee:

- `NULL` on failure, so check with `assert`
- Memory is contiguous; it is not recycled unless you call free
- `realloc` preserves existing data
- `calloc` zero-initializes bytes, `malloc` and `realloc` do not

Undefined behavior occurs:

- If you overflow (i.e., you access beyond bytes allocated)
- If you use after `free`, or if `free` is called twice on a location.
- If you `realloc`/`free` non-heap address

Engineering principles: stack vs heap

Stack ("local variables")

- **Fast**
  Fast to allocate/deallocate; okay to oversize
- **Convenient.**
  Automatic allocation/ deallocation;
  declare/initialize in one step
- **Reasonable type safety**
  Thanks to the compiler

⚠️ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠️ **Somewhat inflexible**
Cannot add/resize at runtime, scope dictated by control flow in/out of functions

Heap (dynamic memory)

Engineering principles: stack vs heap

Stack ("local variables")

- **Fast**
  Fast to allocate/deallocate; okay to oversize
- **Convenient.**
  Automatic allocation/ deallocation; declare/initialize in one step
- **Reasonable type safety**
  Thanks to the compiler

⚠️ **Not especially plentiful**
Total stack size fixed, default 8MB

⚠️ **Somewhat inflexible**
Cannot add/resize at runtime, scope dictated by control flow in/out of functions

Heap (dynamic memory)

- **Plentiful.**
  Can provide more memory on demand!
- **Very flexible.**
  Runtime decisions about how much/when to allocate, can resize easily with realloc
- **Scope under programmer control**
  Can precisely determine lifetime

⚠️ **Lots of opportunity for error**
Low type safety, forget to allocate/free before done, allocate wrong size, etc., Memory leaks (much less critical)

Stack and Heap

- Generally, unless a situation requires dynamic allocation, stack allocation is preferred. Often both techniques are used together in a program.
- Heap allocation is a necessity when:
  - you have a very large allocation that could blow out the stack
  - you need to control the memory lifetime, or memory must persist outside of a function call
  - you need to resize memory after its initial allocation

Recap

- Other heap allocations
- `realloc`

Next time: C Generics – `void *`