CS107, Lecture 11
The Heap, Continued

Reading: K&R 5.6-5.9 or Essential C section 6 on the heap

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan and others.
NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

CS107 Topic 3

How can we effectively manage all types of memory in our programs?

Why is answering this question important?

- Shows us how we can pass around data efficiently with pointers (last time)
- Introduces us to the heap and allocating memory that we manually manage (today)
- Helps us better understand use-after-free vulnerabilities, a common bug (next time)

assign3: implement a function using resizable arrays to read lines of any length from a file and write 2 programs using that function to print the last N lines of a file and print just the unique lines of a file. These programs emulate the **tail** and **uniq** Unix commands!

Learning Goals

- Learn about the differences between the stack and the heap and when to use each one
- Understand how to use the **realloc** function to resize heap allocations later
- Practice using **malloc**, **realloc** and **free** to manage memory on the heap

Announcements

- Assign1 grades released shortly
- Assign3 released today
  - For debugging questions, Helper hours signups starting with assign3 require specific information about what info you've gathered so far – e.g. what have you observed so far with GDB? What's the smallest reproducible test case you've found? Etc.
  - Please also ask us any questions about how to utilize GDB and Valgrind, we're happy to help!

Debugging Check-In

Let us know how you're feeling so far!

Respond on PollEv:
pollev.com/cs107

Lecture Plan

- **Recap:** The Heap So Far
- **Practice:** Pointers, Heap and Valgrind
- **realloc**

```
cp -r /afs/ir/class/cs107/lecture-code/lect11 .
```

Lecture Plan

- **Recap: The Heap So Far**
- **Practice:** Pointers, Heap and Valgrind
- **realloc**

```
cp -r /afs/ir/class/cs107/lecture-code/lect11 .
```

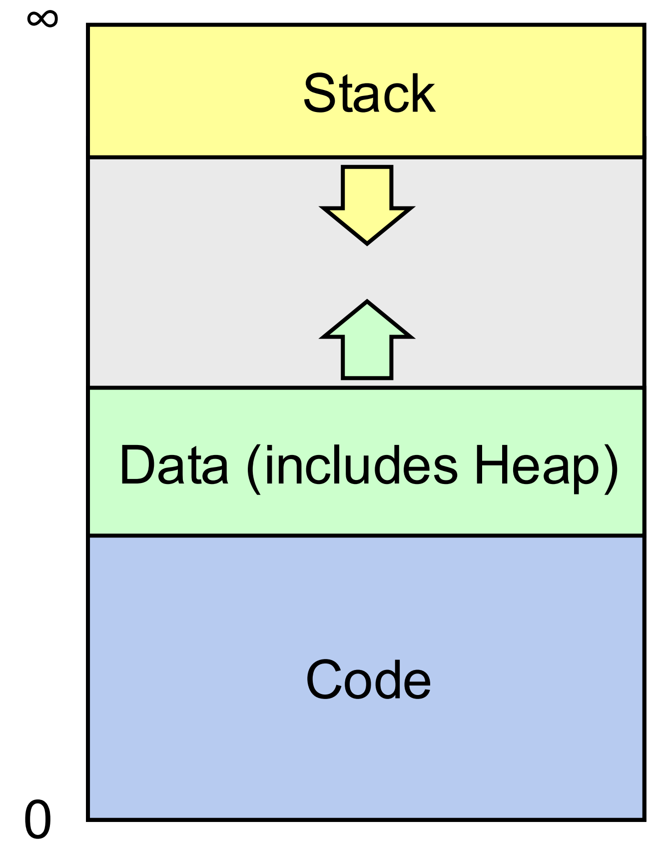
Heap Allocation

The **heap** is an area of memory separate from the stack that we manage ourselves.

Working with the heap consists of 3 core steps:

1. Allocate memory with malloc/realloc/strdup/calloc
2. Assert heap pointer is not NULL
3. Free when done

The heap is **dynamic memory**, so you may encounter many **runtime errors**, even if your code compiles!



1. Allocating Memory

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

malloc returns a pointer to the starting address of a new heap-allocation of the requested size, in bytes, or NULL if there is insufficient space. The memory is not zeroed out.

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

calloc is like malloc but **zeros out** the memory for you. It takes two parameters, which are multiplied to calculate the number of bytes (nmemb * size). **calloc** is more expensive, so use only when zeroing out is important.

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

strdup is a convenience function that returns a **null-terminated**, heap-allocated string with the provided text (like **malloc** + **strcpy**).

2. Always assert with the heap

```
int *arr = malloc(sizeof(int) * 4);
assert(arr != NULL);
```

If an allocation error occurs (e.g. out of heap memory!), malloc/calloc/strdup/etc. will return NULL. This is an important case to check **for robustness**.

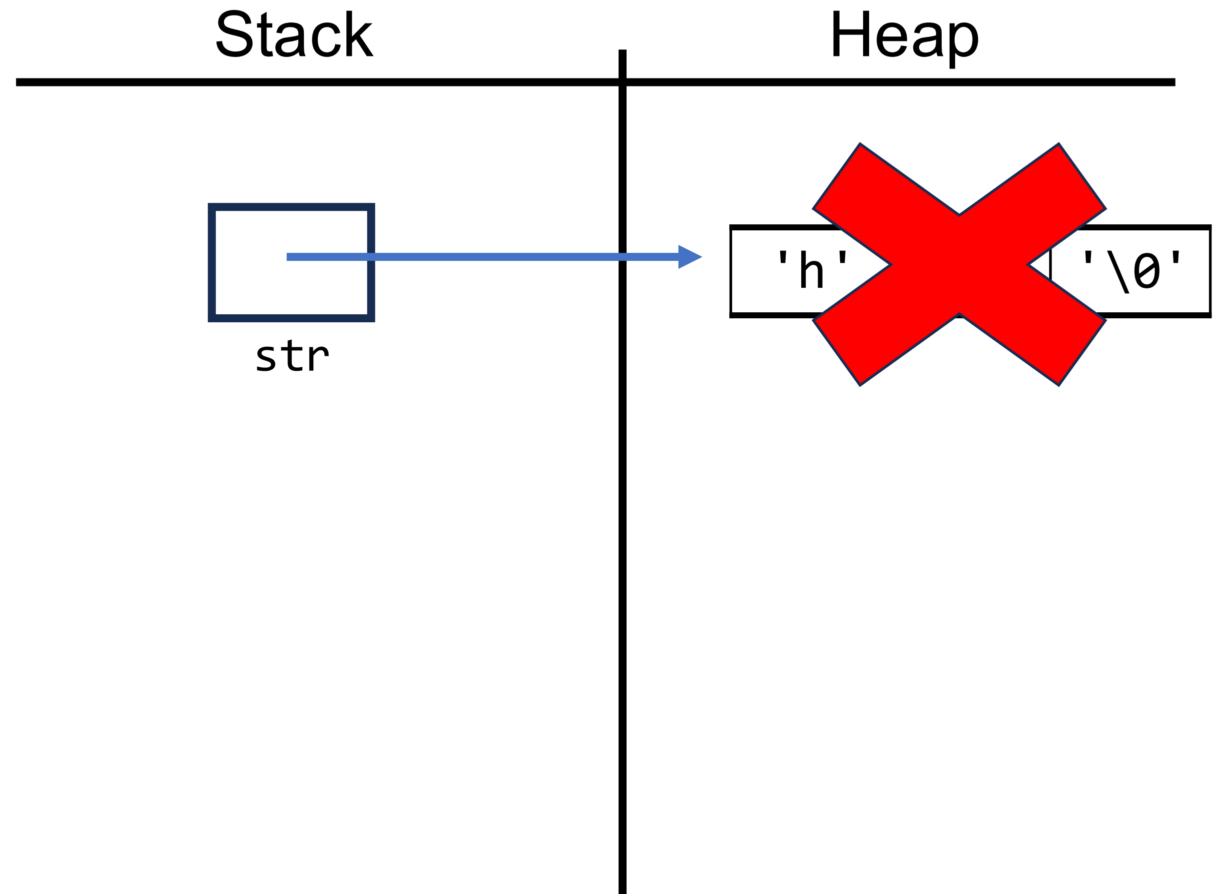
assert will crash the program if the provided condition is false. A memory allocation error is significant, and we should terminate the program.

3. Cleaning Up with `free`

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

free frees the heap memory that **ptr** points to. **ptr** must be a pointer to the start of an allocated block of heap memory. We can continue using **ptr** to point to something else.

```
char *str = strdup("hi");
...
free(str);
str = strdup("107");
```

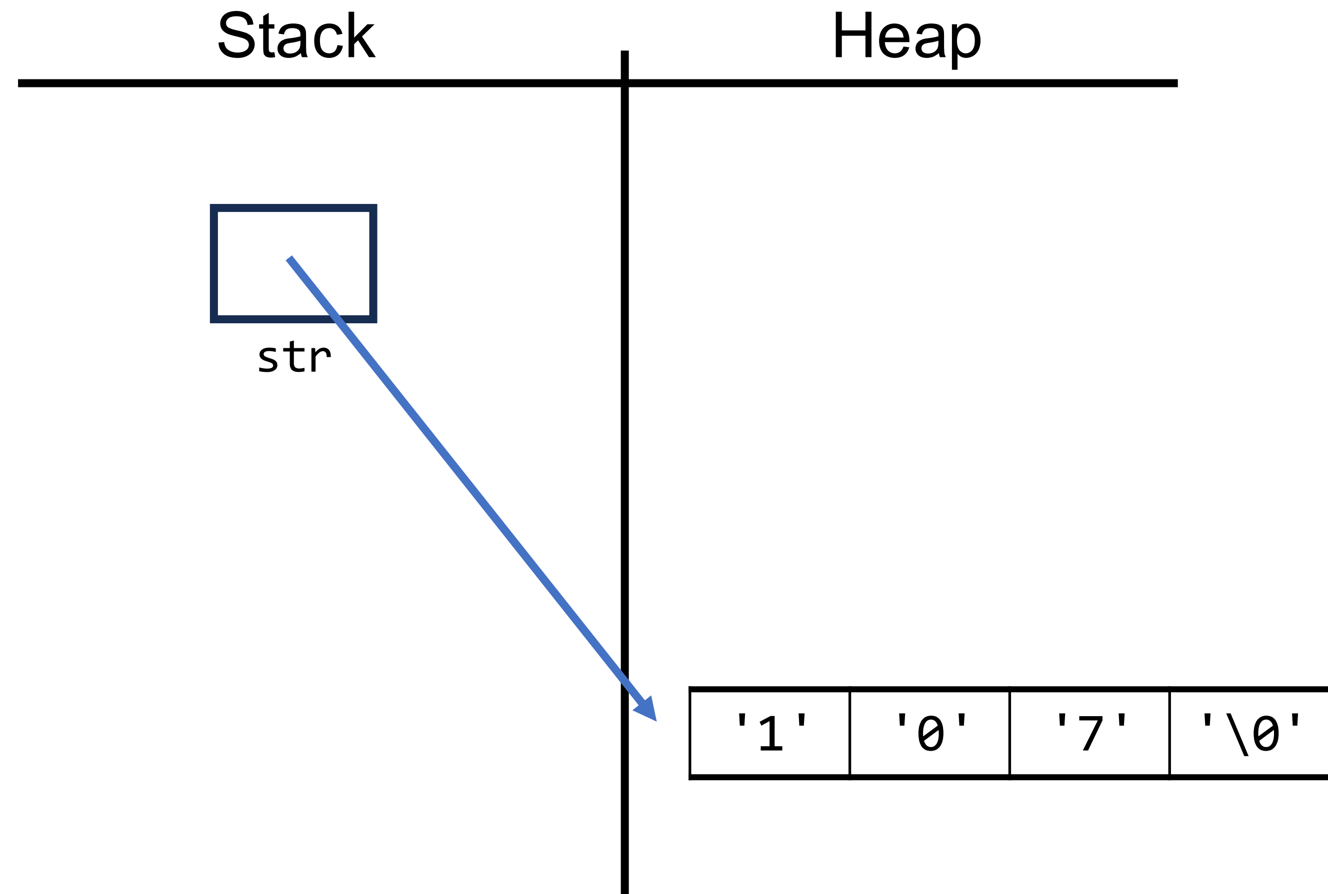


Free

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

free frees the heap memory that **ptr** points to. **ptr** must be a pointer to the start of an allocated block of heap memory. We can continue using **ptr** to point to something else.

```
char *str = strdup("hi");
...
free(str);
str = strdup("107");
```

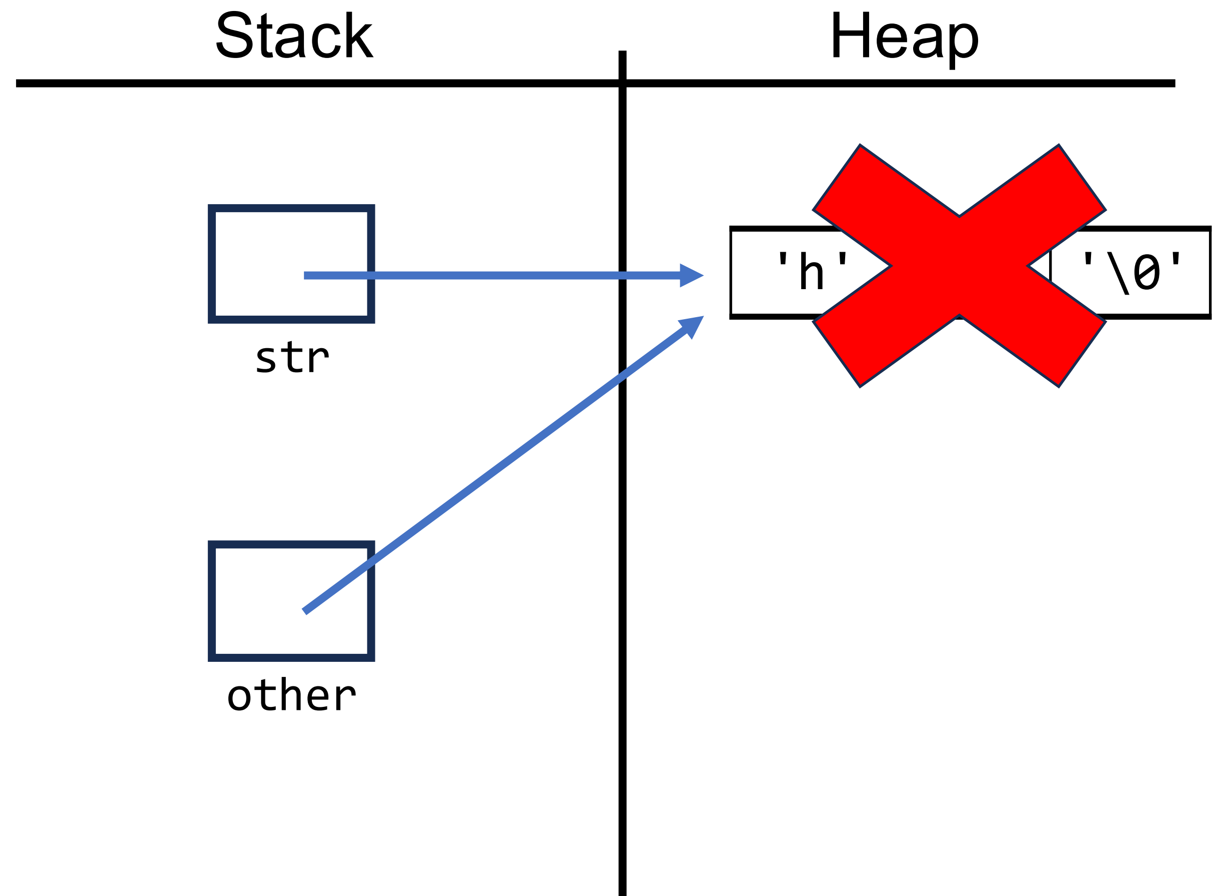


Free

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

We can call free with any pointer pointing to a heap allocation:

```
char *str = strdup("hi");
char *other = str;
// or free(str), but not both!
free(other);
// now str/other should both be
// changed before using
```



Memory Leaks

A **memory leak** is when you do not free memory you previously allocated.

- Your program should be responsible for cleaning up any memory it allocates but no longer needs.
- Tip: free as soon as you are done with a block of memory
- If you never free any memory and allocate an extremely large amount, you may run out of memory in the heap! It can also mean less memory for other programs.
- <u>However, memory leaks rarely (if ever) cause crashes.</u>
- 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!

Lecture Plan

- **Recap:** The Heap So Far
- **Practice: Pointers, Heap and Valgrind**
- **realloc**

```
cp -r /afs/ir/class/cs107/lecture-code/lect11 .
```

Example: Array Compression

Let's implement a function **compress_in_batches** as follows:

```
int *compress_in_batches(const int *values, int n, int *out_n);
```

Returns heap-allocated "compressed" version of provided int array (caller must free it).

- If **array** length <= SIZE_THRESHOLD (declared constant), merge (sum) pairs
- Otherwise, merge (sum) trios of elements
- Store compressed array length at location pointed to by **out_n**
- If array is empty, or not multiple of merge size, returns NULL

Good use-case for heap allocation and pointers – array size not known until function runs, and we can return the array and give back the size as well.

Example: Pig Latin

We will also see an example of how to uncover memory leaks using Valgrind.

valgrind --leak-check=full --show-leak-kinds=all [program info here]

Demo: Array Compression

batch_compression.c

Memory Leaks vs. Memory Errors

Memory Leak: we didn't free all heap memory we allocated. Rarely causes functionality issues, can generally be resolved after program is functionally complete.

Memory Error: we used memory in an invalid way (accessing memory that doesn't belong to us, read uninitialized memory, etc.). Can cause gnarly functionality issues, make sure to resolve immediately!

Lecture Plan

- **Recap:** The Heap So Far
- **Practice:** Pointers, Heap and Valgrind
- **realloc**

```
cp -r /afs/ir/class/cs107/lecture-code/lect11 .
```

realloc

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

The **realloc** function takes an existing allocation pointer and enlarges to a new requested size in bytes. It returns the new pointer. Existing data is preserved.

```
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

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

The **realloc** function takes an existing allocation pointer and enlarges to a new requested size in bytes. **It returns the new pointer.** Existing data is preserved.

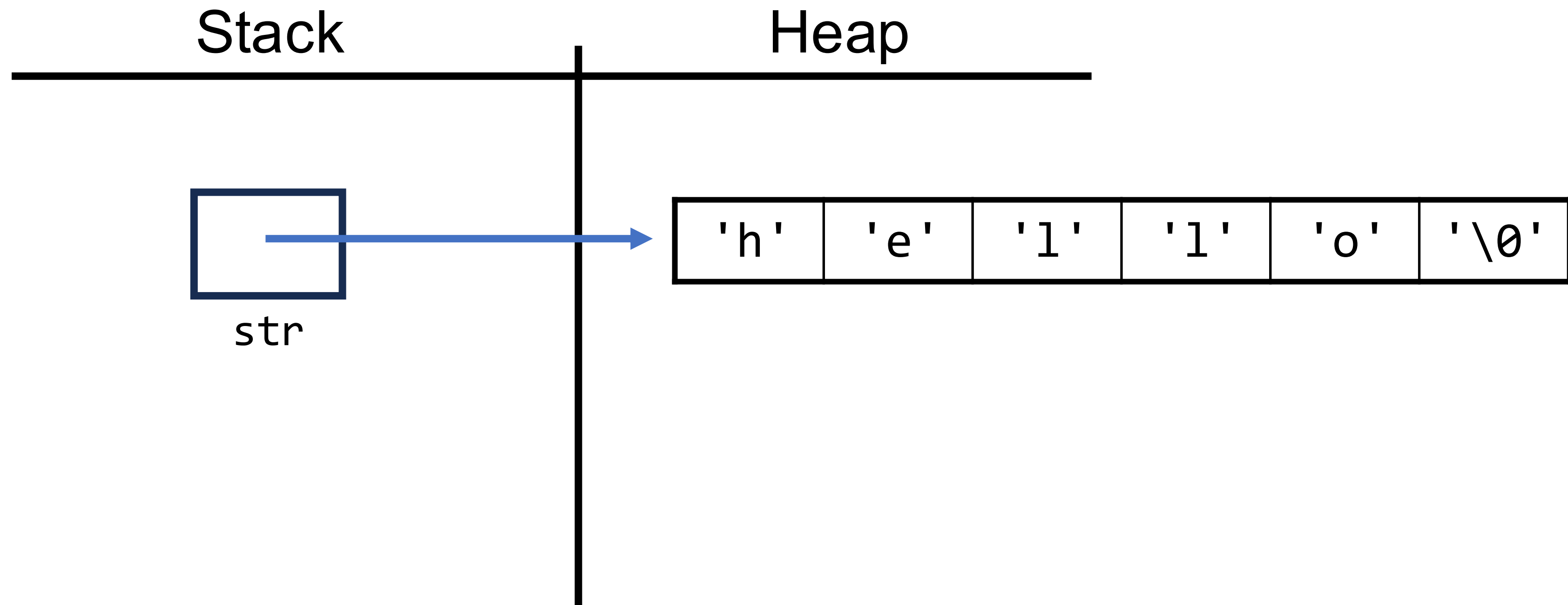
```
char *str = strdup("Hello");
assert(str != NULL);
…

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

Make sure to always save the return value of realloc. Realloc tries to resize the memory in place (if there's empty space after it), but if that's not possible it *moves the memory to a larger location*, frees the old memory, and *returns a pointer to the new location*.

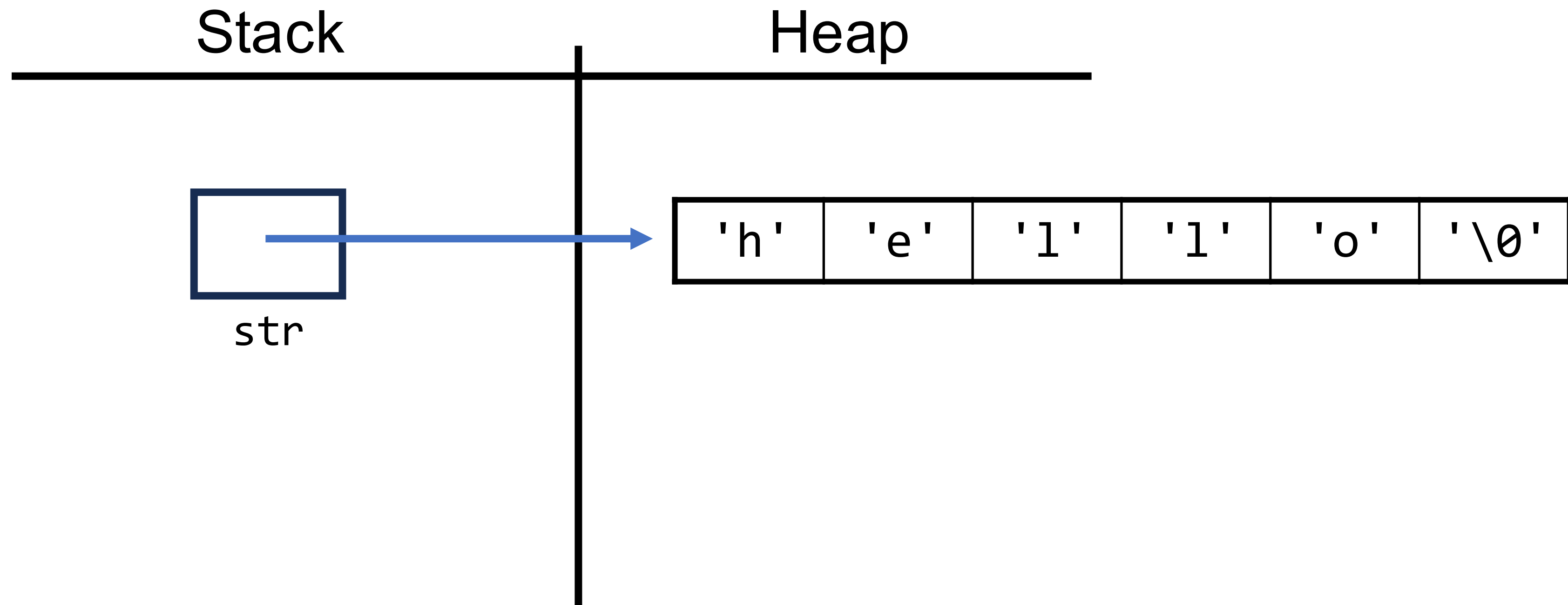
realloc

```
char *str = strdup("Hello");
assert(str != NULL);
…
char *addition = "!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
```



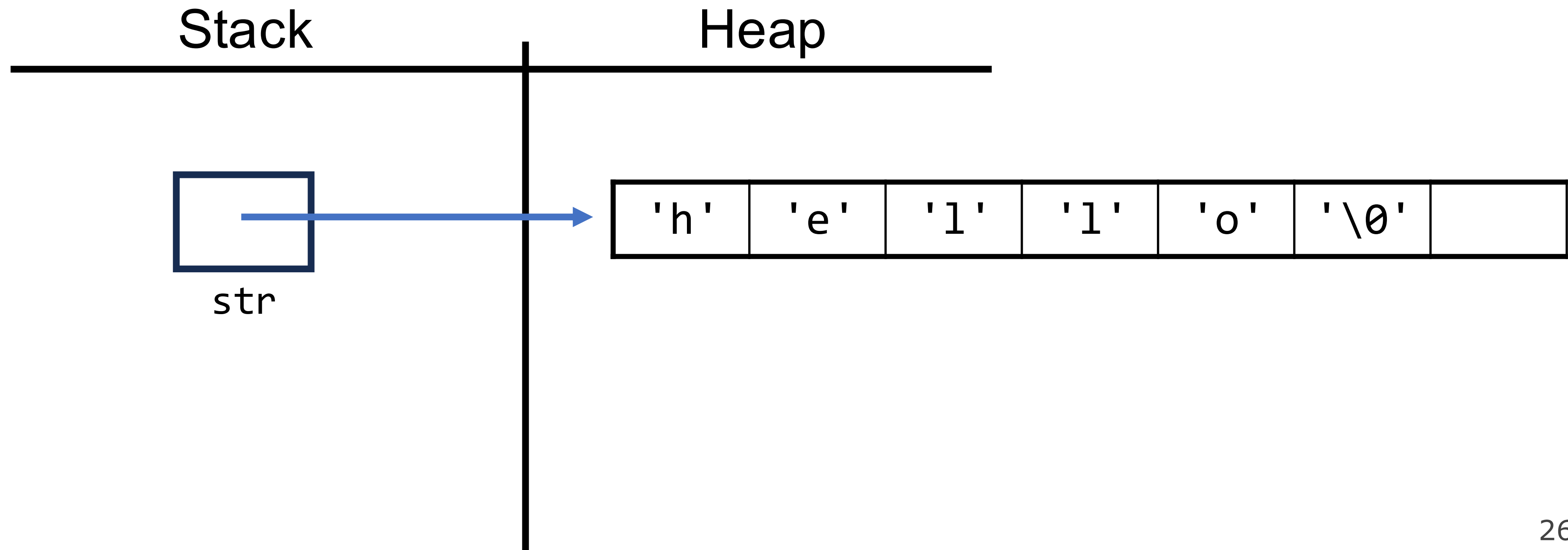
realloc – In Place

```
char *str = strdup("Hello");
assert(str != NULL);
…
char *addition = "!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
```



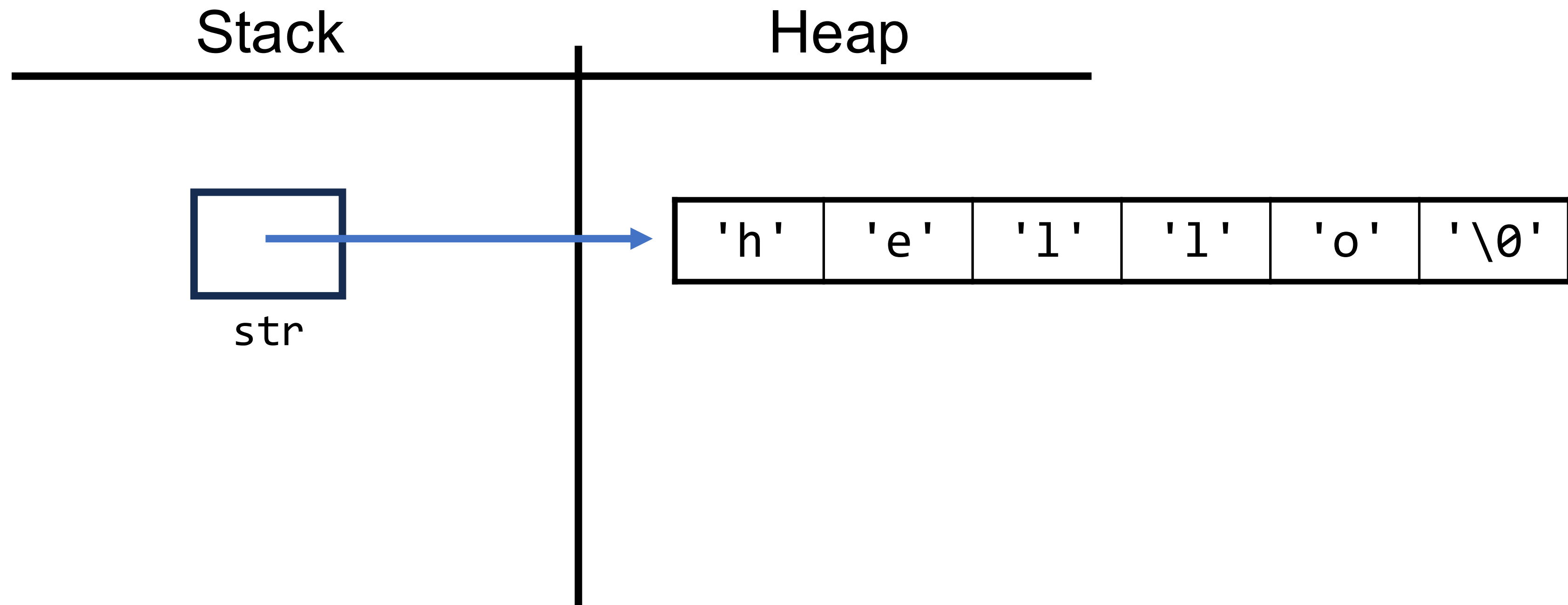
realloc – In Place

```
char *str = strdup("Hello");
assert(str != NULL);
…
char *addition = "!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
```



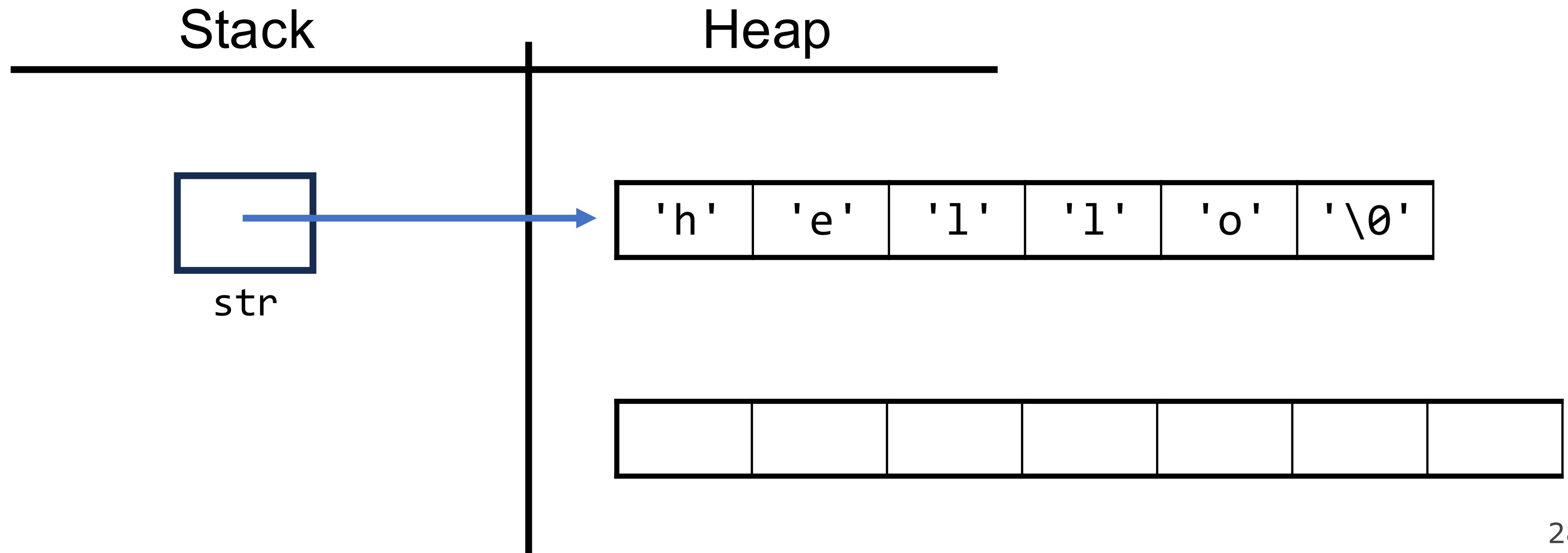
realloc – Moving

```
char *str = strdup("Hello");
assert(str != NULL);
…
char *addition = "!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
```



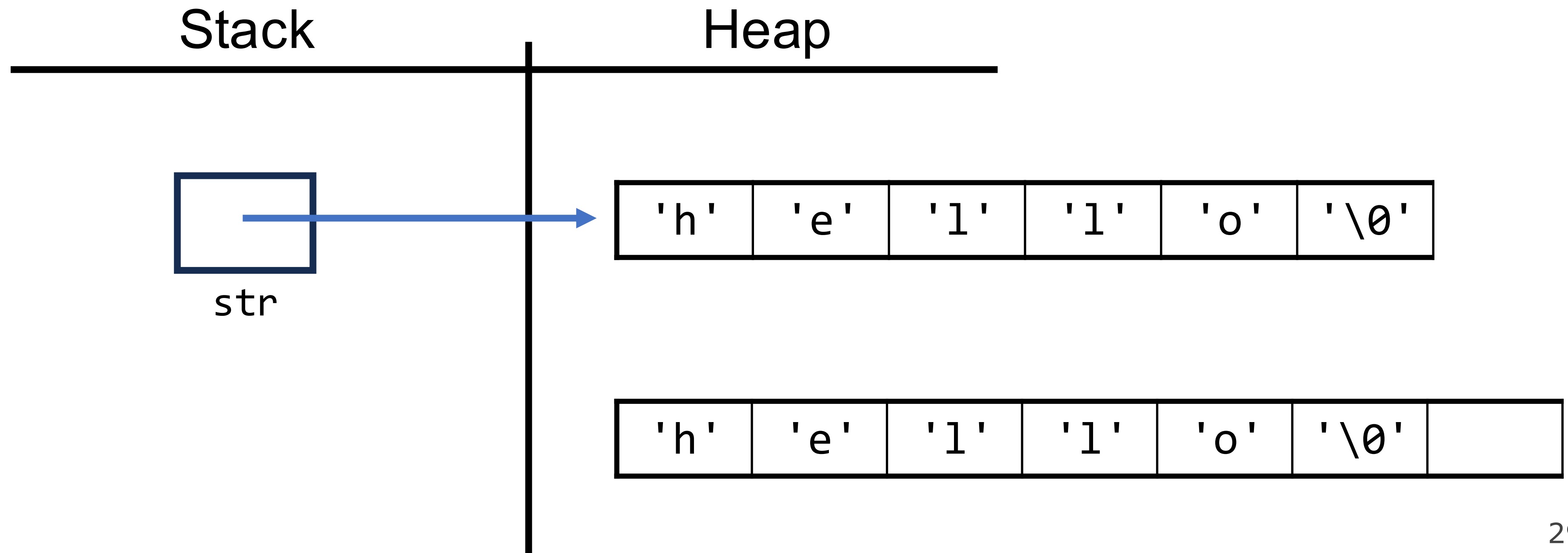
realloc

```
char *str = strdup("Hello");
assert(str != NULL);
…
char *addition = "!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
```



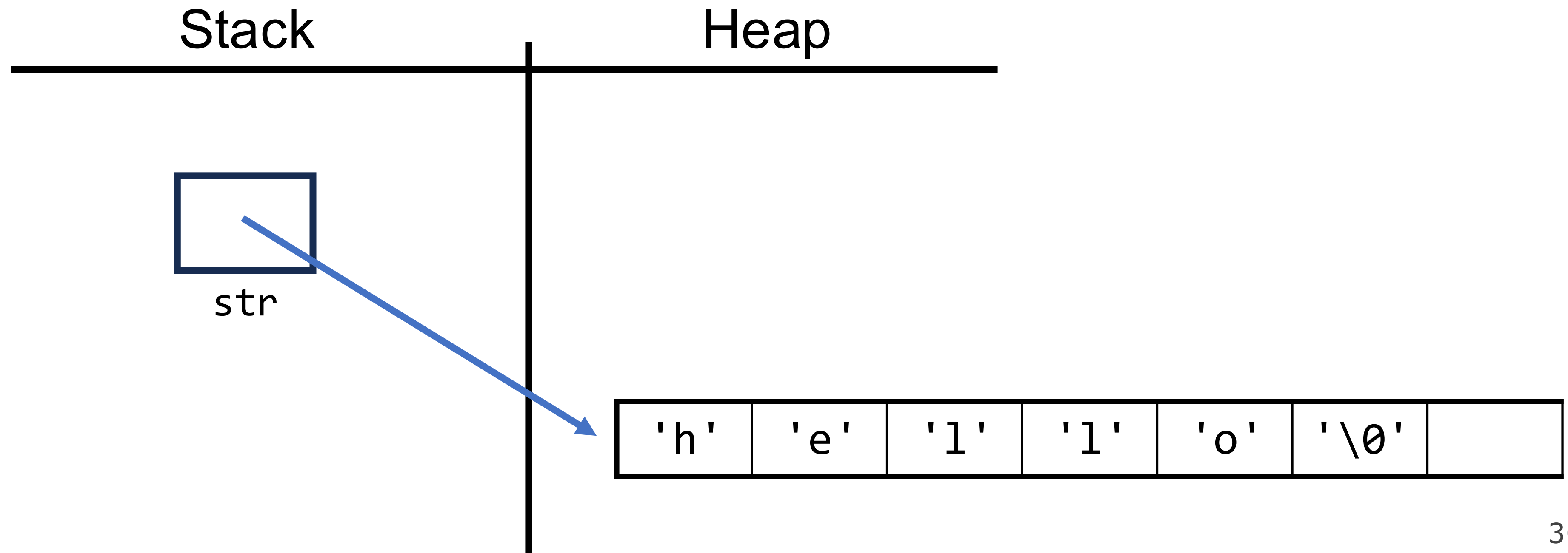
realloc

```
char *str = strdup("Hello");
assert(str != NULL);
…
char *addition = "!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
```



realloc

```
char *str = strdup("Hello");
assert(str != NULL);
…
char *addition = "!";
str = realloc(str, strlen(str) + strlen(addition) + 1);
```

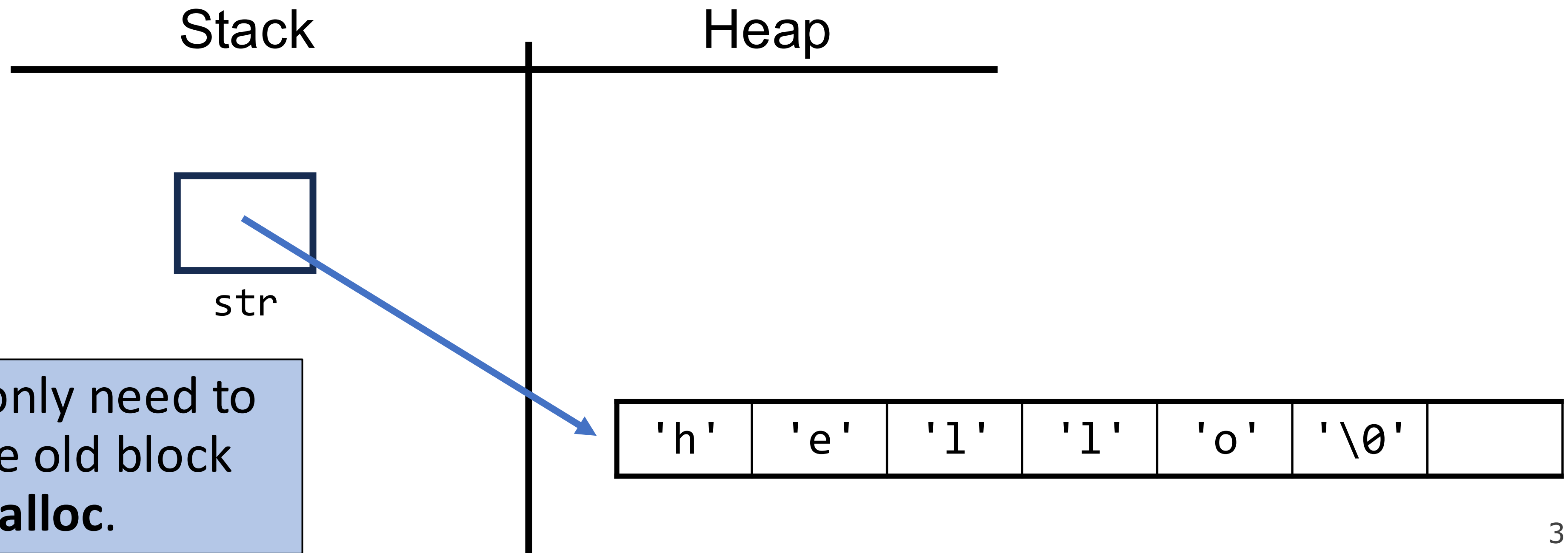


realloc

```
char *str = strdup("Hello");
assert(str != NULL);
…
char *addition = "!";
str = realloc(str, strlen(str) + strlen(addition) + 1);

…

free(str);
```



If we were moved, we only need to free the new block – the old block was already freed by **realloc**.

realloc

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

realloc only accepts pointers that were previously returned by malloc/etc. heap-allocation functions – cannot pass stack pointers or pointers to the middle of a heap-allocate block.

Exercise: `realloc`

Let's write code that uses realloc to double the size of an int array:

```
1 int len = ...
2 int *arr = malloc(sizeof(int) * len);
3 assert(arr != NULL);
4 ...
5 // we want to double the size of arr
6 arr = realloc(arr, /* TODO: what goes here? */ );
```

Line 6: What should the second argument to `realloc` be?

A. `len`
B. `len * 2`
C. `len * sizeof(int)`
D. `(len * 2) * sizeof(int)`

Respond on PollEv:
pollev.com/cs107

L11. 2. What should the second parameter to realloc be?

len

0%

len * 2

0%

len * sizeof(int)

0%

(len * 2) * sizeof(int)

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Exercise: `realloc`

Let's write code that uses realloc to double the size of an int array:

```
1 int len = ...
2 int *arr = malloc(sizeof(int) * len);
3 assert(arr != NULL);
4 ...
5 // we want to double the size of arr
6 arr = realloc(arr, /* TODO: what goes here? */ );
```

Line 6: What should the second argument to `realloc` be?

A. `len`
B. `len * 2`
C. `len * sizeof(int)`
D. `(len * 2) * sizeof(int)`

Key idea: **realloc**'s second parameter must be the total size of the new allocation in bytes.

Demo: Data Reading

`streaming_data.c`

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

- **Recap:** The Heap So Far
- **Practice:** Pointers, Heap and Valgrind
- realloc

Next time: C Generics

Lecture 11 takeaway: We can allocate memory on the heap to manage it ourselves. We manipulate heap memory via pointers. We must free memory when we are done with it, and can resize prior allocations with realloc.

Structs

Structs

A *struct* is a way to define a new variable type that is a group of other variables.

```
struct date {            // declaring a struct type
    int month;
    int day;             // members of each date structure
};
…

struct date today;                             // construct structure instances
today.month = 1;
today.day = 28;

struct date new_years_eve = {12, 31};   // shorter initializer syntax
```

Structs

Wrap the struct definition in a **typedef** to avoid having to include the word **struct** every time you make a new variable of that type.

```
typedef struct date {
    int month;
    int day;
} date;
…

date today;
today.month = 1;
today.day = 28;

date new_years_eve = {12, 31};
```

Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct.

```
void advance_day(date d) {
	d.day++;
}

int main(int argc, char *argv[]) {
	date my_date = {1, 28};
	advance_day(my_date);
	printf("%d", my_date.day); // 28
	return 0;
}
```

Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct. **Use a pointer to modify a specific instance.**

```
void advance_day(date *d) {
	(*d).day++;
}

int main(int argc, char *argv[]) {
	date my_date = {1, 28};
	advance_day(&my_date);
	printf("%d", my_date.day); // 29
	return 0;
}
```

Structs

The **arrow** operator lets you access the field of a struct pointed to by a pointer.

```
void advance_day(date *d) {
      d->day++;          // equivalent to (*d).day++;
}

int main(int argc, char *argv[]) {
      date my_date = {1, 28};
      advance_day(&my_date);
      printf("%d", my_date.day); // 29
      return 0;
}
```

Structs

C allows you to return structs from functions as well. It returns whatever is contained within the struct.

```
date create_new_years_date() {
     date d = {1, 1};
     return d;          // or return (date){1, 1};
}

int main(int argc, char *argv[]) {
     date my_date = create_new_years_date();
     printf("%d", my_date.day); // 1
     return 0;
}
```

Structs

sizeof gives you the entire size of a struct, which is the sum of the sizes of all its contents.

```
typedef struct date {
      int month;
      int day;
  } date;

int main(int argc, char *argv[]) {
      int size = sizeof(date);    // 8
      return 0;
}
```

Arrays of Structs

You can create arrays of structs just like any other variable type.

```
typedef struct my_struct {
    int x;
    char c;
} my_struct;

…

my_struct array_of_structs[5];
```

Arrays of Structs

To initialize an entry of the array, you must use this special syntax to confirm the type to C.

```
typedef struct my_struct {
    int x;
    char c;
} my_struct;

…

my_struct array_of_structs[5];
array_of_structs[0] = (my_struct){0, 'A'};
```

Arrays of Structs

You can also set each field individually.

```
typedef struct my_struct {
    int x;
    char c;
} my_struct;

…
my_struct array_of_structs[5];
array_of_structs[0].x = 2;
array_of_structs[0].c = 'A';
```

Extra Practice

Goodbye, Free Memory

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

```
1 char *str = strdup("Hello");
2 assert(str != NULL);
3 char *ptr = str + 1;
4 for (int i = 0; i < 5; i++) {
5   int *num = malloc(sizeof(int));
6   *num = i;
7   printf("%s %d\n", ptr, *num);
8 }
9 printf("%s\n", str);
```

Recommendation: Don’t worry about putting in frees until **after** you’re finished with functionality.

Memory leaks will rarely crash your CS107 programs.

Goodbye, Free Memory

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

```
1  char *str = strdup("Hello");
2  assert(str != NULL);
3  char *ptr = str + 1;
4  for (int i = 0; i < 5; i++) {
5    int *num = malloc(sizeof(int));
6    *num = i;
7    printf("%s %d\n", ptr, *num);
8    free(num);
9  }
10 printf("%s\n", str);
11 free(str);
```

Recommendation: Don't worry about putting in frees until **after** you're finished with functionality.

Memory leaks will rarely crash your CS107 programs.

strcat_extend

Write a function that takes in a heap-allocated **str1,** enlarges it, and concatenates **str2** onto it.

```
1 char *strcat_extend(char *heap_str, const char *concat_str) {
2     (____________________(1)____________________);
3     heap_str = realloc(___(2A)___,___(2B)___);
4     (____________________(3)____________________);
5     strcat(___(3A)___, ___(3B)___);
6     return heapstr;
7 }
```

Example usage:

```
char *str = strdup("Hello ");
str = strcat_extend(str, "world!");
printf("%s\n", str);
free(str);
```

strcat_extend

Write a function that takes in a heap-allocated **str1,** enlarges it, and concatenates **str2** onto it.

```
1 char *strcat_extend(char *heap_str, const char *concat_str) {
2     int new_length = strlen(heap_str) + strlen(concat_str) + 1;
3     heap_str = realloc(heap_str, new_length);
4     assert(heap_str != NULL);
5     strcat(heap_str, concat_str);
6     return heapstr;
7 }
```

Example usage:

```
char *str = strdup("Hello ");
str = strcat_extend(str, "world!");
printf("%s\n", str);
free(str);
```