

Dynamic Memory & ADTs in C

Optional Textbook Readings: CP:AMA 17.1, 17.2, 17.3, 17.4

The primary goal of this section is to be able to use dynamic memory.

The heap

The *heap* is the final section in the C memory model.

It can be thought of as a big “pile” (or “pool”) of memory that is available to a program.

Memory is **allocated** from the heap upon request.

This memory is “*borrowed*”, and can be “*returned*” (“*freed*”) back to the heap when it is no longer needed (deallocation).

Returned memory may be **reused** (“*recycled*”) for a future allocation.

If too much memory has already been allocated, attempts to borrow additional memory fail.

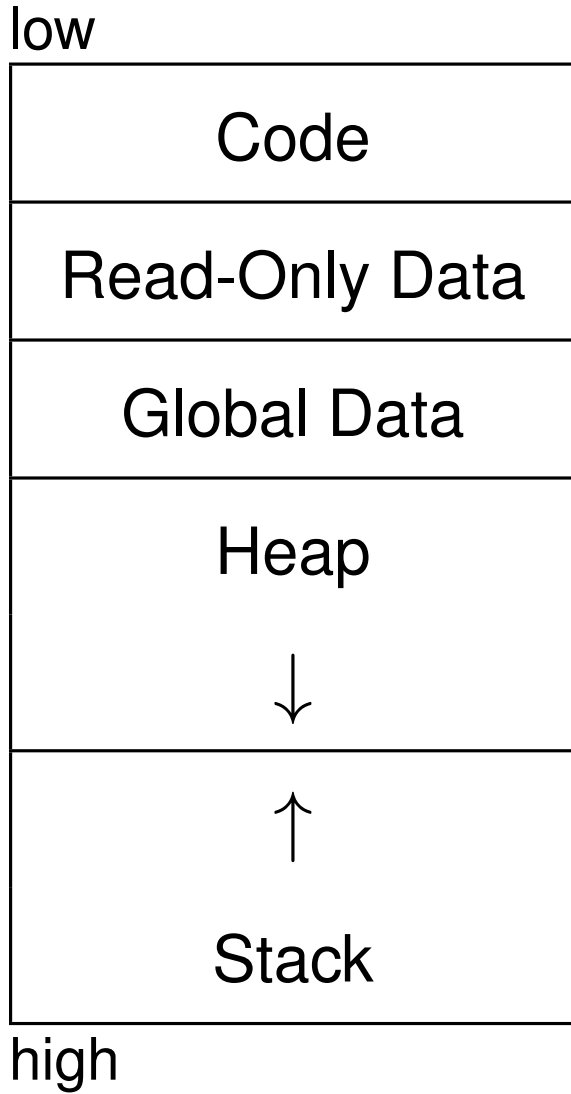
Advantages of heap-allocated memory

Dynamic: The allocation size can be determined at run time (while the program is running).

Resizable: A heap allocation can be “resized”.

Duration: Heap allocations *persist* until they are “freed”. A function can create a data structure (one or more allocations) that continues to exist after the function returns.

Safety: If memory runs out, it can be detected and handled properly (unlike stack overflows).



Unfortunately, there is also a *data structure* known as a heap, and the two are unrelated.

To avoid confusion, prominent computer scientist Donald Knuth campaigned to use the name “free store” or the “memory pool”, but the name “heap” has stuck.

A similar problem arises with “the stack” region of memory because there is also a Stack ADT. However, their behaviour is very similar so it is far less confusing.

malloc

The `malloc` (**m**emory **a**llocation) function obtains memory from the heap. It is provided in `<stdlib.h>`.

```
// malloc(s) requests s bytes of contiguous memory from the heap
//   and returns a pointer to a block of s bytes, or
//   NULL if not enough contiguous memory is available
// time: O(1) [close enough for this course]
```

For example, for an array of 100 ints:

```
int *my_array = malloc(100 * sizeof(int));
```

or a single struct `posn`:

```
struct posn *my_posn = malloc(sizeof(struct posn));
```

These two examples illustrate the most common use of dynamic memory: allocating space for **arrays and structures**.

Uninitialized

The heap memory provided by `malloc` is **uninitialized**.

```
struct posn *my_posn = malloc(sizeof(struct posn));  
  
printf("mystery values: (%d, %d)\n", my_posn->x,  
                                     my_posn->y);
```

There is also a `calloc` function which essentially calls `malloc` and then “initializes” the memory by filling it with zeros. `calloc` is $O(n)$, where n is the size of the block.

Dynamic Arrays

With pointer notation, the syntax for accessing a heap array is nearly identical to the syntax for a stack (or global) array.

```
int *my_array = malloc(n * sizeof(int));

// initialize the array
for (int i = 0; i < n; ++i) {
    my_array[i] = 0;
}
```

We could use `malloc(n * 4)`, but `malloc(n * sizeof(int))` is clearer and better style.

Always use `sizeof` with `malloc` to improve communication.

Strictly speaking, the type of the `malloc` parameter is `size_t`, which is a special type produced by the `sizeof` operator.

`size_t` and `int` are different types of integers.

`Seashe11` is mostly forgiving, but in other C environments using an `int` when C expects a `size_t` may generate a warning.

The proper `printf` format specifier to print a `size_t` is `%zd`.

The declaration for the `malloc` function is:

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

The return type is a `(void *)` (*void pointer*), a special pointer that can point at *any* type.

```
int *my_array = malloc(100 * sizeof(int));
```

```
struct posn *my_posn = malloc(sizeof(struct posn));
```

For this course, we assume `malloc` is $O(1)$.

In practice, the constant time may be quite large.

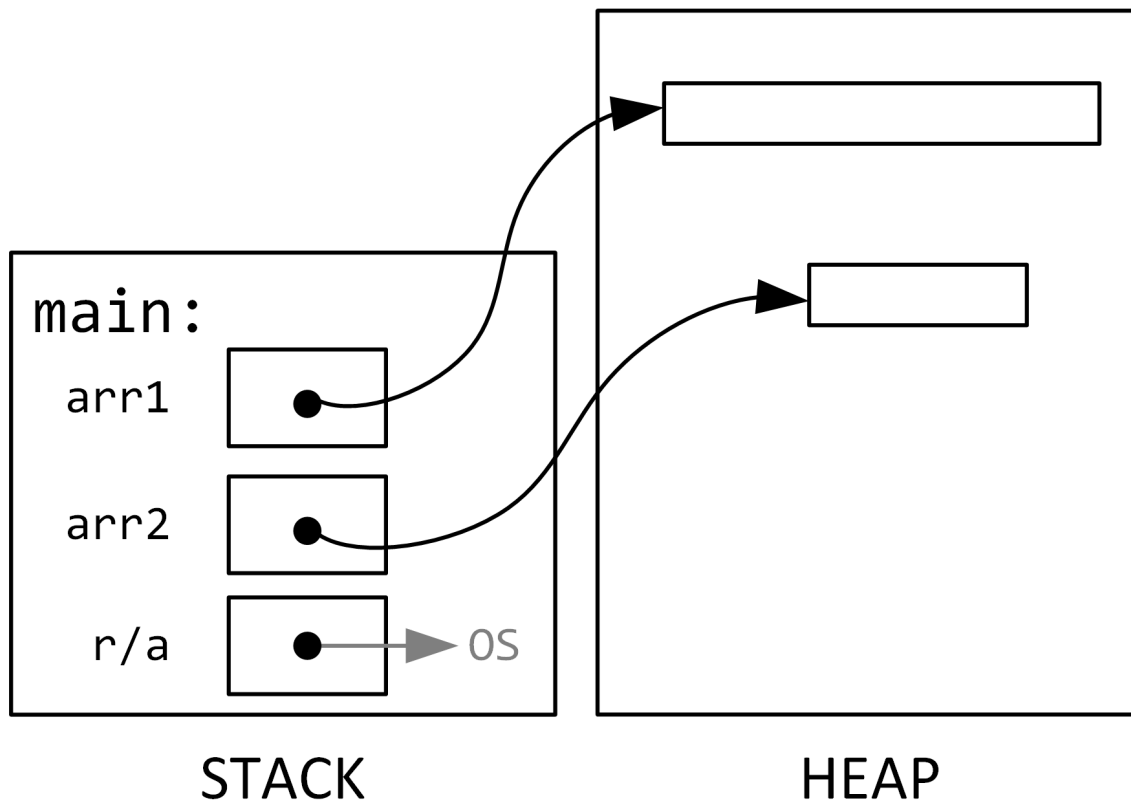
In addition, the running time for `malloc` depends on the state of the heap.

After multiple allocations and deallocations, the heap can become *fragmented*, which may affect the performance of `malloc`.

This is one of the disadvantages of heap memory.

example: visualizing the heap

```
int main(void) {  
    int *arr1 = malloc(10 * sizeof(int));  
    int *arr2 = malloc(5 * sizeof(int));  
    //...  
}
```



Out of memory

An unsuccessful call to `malloc` returns `NULL`.

In practice it's good style to check every `malloc` return value and gracefully handle a `NULL` instead of crashing.

```
int *my_array = malloc(n * sizeof(int));
if (my_array == NULL) {
    // handle out of memory
    // ...
}
```

In the “real world” you should **always** perform this check, but in this course, you do **not** have to check for a `NULL` return value unless instructed otherwise.

In these notes, we omit this check to save space.

free

Every block of memory obtained through `malloc` must eventually be freed (when the memory is no longer in use).

```
// free(p) returns memory at p back to the heap
// requires: p must be from a previous malloc
// effects:  the memory at p is invalid
// time:    0(1)
```

In the Seashell environment, every block **must** be freed.

```
int *my_array = malloc(n * sizeof(int));
// ...
// ...
free(my_array);
```

Dangling pointers

Once a block of memory is freed, accessing that memory is invalid and may cause errors (or unpredictable results).

Consider this example:

```
int *my_array = malloc(10 * sizeof(int));  
free(my_array);
```

The memory `my_array` is pointing at has been freed and is now invalid, but `my_array` is **still pointing at it**.

`free(my_array)` does **not** modify the pointer `my_array`.

A pointer to a freed allocation is known as a “*dangling pointer*”. It is often good style to assign `NULL` to a dangling pointer.

Invalid behaviour with free

It is invalid to free memory that was not returned by a `malloc` or that has already been freed. `free(NULL)` is okay and simply ignored.

```
int *my_array = malloc(10 * sizeof(int));

free(my_array + 1);    // INVALID

free(my_array);       // VALID (my_array is now dangling)

int i = my_array[0];  // INVALID
my_array[0] = 42;     // INVALID

free(my_array);       // INVALID

my_array = NULL;      // GOOD STYLE (no longer dangling)
free(my_array);       // IGNORED
```


Memory leaks

A memory leak occurs when allocated memory is not eventually freed.

Programs that leak memory may suffer degraded performance or eventually crash.

```
int *my_array = NULL;
my_array = malloc(10 * sizeof(int));
my_array = malloc(10 * sizeof(int)); // Memory Leak!
```

In this example, the address from the original `malloc` has been overwritten.

That memory is now *“lost”* (or *leaked*) and so it can never be freed.

Garbage collection

Many modern languages (including Racket) have a ***garbage collector***.

A garbage collector **detects** when memory is no longer in use and **automatically** frees memory and returns it to the heap.

One disadvantage of a garbage collector is that it can be slow and affect performance, which is a concern in high performance computing.

C does not have a garbage collector, so you will have to ensure your programs have no memory leaks.

Merge Sort

In *merge sort*, the array is split (in half) into two separate arrays. The two arrays are sorted and then they are merged back together into the original array.

This is another example of a *divide and conquer* algorithm.

The arrays are *divided* into two smaller problems, which are then sorted (*conquered*). The results are combined to solve the original problem.

To simplify our implementation, we use a merge helper function.

```
// merge(dest, src1, len1, src2, len2) modifies dest to contain
//   the elements from both src1 and src2 in sorted order
// requires: length of dest is at least (len1 + len2)
//           src1 and src2 are sorted [not asserted]
// effects: modifies dest
// time:    0(n), where n is len1 + len2
```

```
void merge(int dest[], const int src1[], int len1,
           const int src2[], int len2) {
    int pos1 = 0;
    int pos2 = 0;
    for (int i = 0; i < len1 + len2; ++i) {
        if (pos1 == len1 || (pos2 < len2 && src2[pos2] < src1[pos1])) {
            dest[i] = src2[pos2];
            ++pos2;
        } else {
            dest[i] = src1[pos1];
            ++pos1;
        }
    }
}
```

```

void merge_sort(int a[], int len) {
    if (len <= 1) return;
    int llen = len / 2;
    int rlen = len - llen;

    int *left = malloc(llen * sizeof(int));
    int *right = malloc(rlen * sizeof(int));

    for (int i = 0; i < llen; ++i) left[i] = a[i];
    for (int i = 0; i < rlen; ++i) right[i] = a[i + llen];

    merge_sort(left, llen);
    merge_sort(right, rlen);

    merge(a, left, llen, right, rlen);

    free(left);
    free(right);
}

```

Merge sort is $O(n \log n)$, even in the *worst case*.

Duration

A significant advantage of dynamic memory is that a function can obtain memory that **persists after** the function has returned.

```
// build_array(len) returns a new array initialized with
//   values a[0] = 0, a[1] = 1, ... a[len-1] = len-1
// requires: len > 0
// effects: allocates memory (caller must free)
```

```
int *build_array(int len) {
    assert(len > 0);
    int *a = malloc(len * sizeof(int));
    for (int i = 0; i < len; ++i) {
        a[i] = i;
    }
    return a;    // array exists beyond function return
}
```

Dynamic memory side effect

Allocating (and deallocating) memory has a **side effect**: it modifies the “state” of the heap.

A function that allocates persistent memory (*i.e.*, not freed) has a side effect and must be documented.

The caller (client) is responsible for freeing the memory (communicate this).

```
// build_array(n) returns a new array...  
// effects: allocates memory (caller must free)
```

A function could also free memory it did not allocate.

That would also be a side effect:

```
// process_and_destroy_array(a, len) ...  
// requires: a is a heap-allocated array  
// effects:  frees a (a is now invalid)
```

This behaviour is rare outside of ADTs.

example: strdup

The `strdup` function makes a duplicate (copy) of a string.

```
// my_strdup(s) makes a duplicate of s
// effects: allocates memory (caller must free)

char *my_strdup(const char *s) {
    char *newstr = malloc((strlen(s) + 1) * sizeof(char));
    strcpy(newstr, s);
    return newstr;
}
```

Recall that the `strcpy(dest, src)` copies the characters from `src` to `dest`, and that the `dest` array must be large enough.

When allocating memory for strings, don't forget to include space for the null terminator.

Resizing arrays

Because `malloc` requires the size of the block of memory to be allocated, it does not seem to solve the problem:

“What if we do not know the length of an array in advance?”

To solve this problem, we can **resize** an array by:

- creating a new array
- copying the items from the old to the new array
- freeing the old array

example: resizing an array

As we will see shortly, this is not how it is done in practice, but this is an illustrative example.

```
// my_array has a length of 100
int *my_array = malloc(100 * sizeof(int));

// ...

// oops, my_array now needs to have a length of 101
int *old = my_array;
my_array = malloc(101 * sizeof(int));
for (int i = 0; i < 100; ++i) {
    my_array[i] = old[i];
}
free(old);
```

realloc

To make resizing arrays easier, there is a `realloc` function.

```
// realloc(p, newsize) resizes the memory block at p
//   to be newsize and returns a pointer to the
//   new location, or NULL if unsuccessful
// requires: p must be from a previous malloc/realloc
//           or NULL (then realloc behaves like malloc)
// effects:  the memory at p is invalid (freed)
// time:     0(n), where n is min(newsize, oldsize)
```

Similar to our previous example, `realloc` preserves the contents from the old array location.

```
int *my_array = malloc(100 * sizeof(int));
// ...
my_array = realloc(my_array, 101 * sizeof(int));
```

The pointer returned by `realloc` may actually be the *original* pointer, depending on the circumstances.

Regardless, after `realloc` **only the new returned pointer can be used.**

Assume that the address passed to `realloc` was freed and is now **invalid**.

Always think of `realloc` as a `malloc`, a “copy”, then a `free`.

Typically, `realloc` is used to request a larger size and the additional memory is *uninitialized*.

If the size is smaller, the extraneous memory is discarded.

Be careful using `realloc` inside of a *helper* function.

```
// repeat(s) modifies s by repeating it ("abc" => "abcabc")
// and returns the new s
// requires: s is a heap-allocated string
// effects: re-allocates memory (s is invalid)
```

```
char *repeat(char *s) {
    int len = strlen(s);
    s = realloc(s, (len * 2 + 1) * sizeof(char));
    for (int i = 0; i < len; ++i) {
        s[i + len] = s[i];
    }
    s[len * 2] = '\0';
    return s;                // this is ESSENTIAL
}
```

A common mistake is to make `repeat` a `void` function (not return the new address for `s`).

This causes a **memory leak** if the address of `s` changes.

Although rare, in practice,

```
my_array = realloc(my_array, newsize);
```

could possibly cause a memory leak if an “out of memory” condition occurs.

In C99, an unsuccessful `realloc` returns `NULL` and the original memory block is not freed.

```
// safer use of realloc
int *tmp = realloc(my_array, newsize);
if (tmp) {
    my_array = tmp;
} else {
    // handle out of memory condition
}
```

String I/O: strings of unknown length

In Section 07 we saw how reading in strings can be susceptible to buffer overruns.

```
char str[81];  
int retval = scanf("%s", str);
```

The target array is often oversized to ensure there is capacity to store the string. Unfortunately, regardless of the length of the array, a buffer overrun may occur.

To solve this problem we can continuously resize (`realloc`) an array while reading in only one character at a time.


```

// read_str_slow() reads in a non-whitespace string from I/O
// or returns NULL if unsuccessful
// effects: allocates memory (caller must free)
// time: O(n^2)

char *read_str_slow(void) {
    char c = 0;
    if (scanf(" %c", &c) != 1) return NULL; // ignore initial WS
    int len = 1;
    char *str = malloc(len * sizeof(char));
    str[0] = c;
    while (1) {
        if (scanf("%c", &c) != 1) break;
        if (c == ' ' || c == '\n') break;
        ++len;
        str = realloc(str, len * sizeof(char));
        str[len - 1] = c;
    }
    str = realloc(str, (len + 1) * sizeof(char));
    str[len] = '\0';
    return str;
}

```

Improving the efficiency

Unfortunately, the running time of `read_str_slow` is $O(n^2)$, where n is the length of the string.

This is because `realloc` is $O(n)$ and occurs inside of the loop.

A better approach might be to allocate **more memory than necessary** and only call `realloc` when the array is “full”.

A popular strategy is to **double** the length of the array when it is full.

Similar to working with *oversized arrays*, we need to keep track of the “*actual*” length in addition to the *allocated* length.

```

// time: O(n) [see analysis on next slide]
char *read_str(void) {
    char c = 0;
    if (scanf(" %c", &c) != 1) return NULL; // ignore initial WS
    int maxlen = 1;
    int len = 1;
    char *str = malloc(maxlen * sizeof(char));
    str[0] = c;
    while (1) {
        if (scanf("%c", &c) != 1) break;
        if (c == ' ' || c == '\n') break;
        if (len == maxlen) {
            maxlen *= 2;
            str = realloc(str, maxlen * sizeof(char));
        }
        ++len;
        str[len - 1] = c;
    }
    str = realloc(str, (len + 1) * sizeof(char));
    str[len] = '\0';
    return str;
}

```

With our “doubling” strategy, most iterations are $O(1)$, unless it is necessary to resize (`realloc`) the array.

The resizing time for the first 32 iterations would be:

1,2,0,4,0,0,0,8,0,0,0,0,0,0,0,16,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,32

For n iterations, the total resizing time is at most:

$$1 + 2 + 4 + \dots + \frac{n}{2} + n = 2n - 1 = O(n).$$

By using this doubling strategy, the **total** run time for `read_str` is now $O(n)$.

Reading in an array of strings

In Section 09, we discussed how an array of strings is often stored as an array of pointers (of type `char **`).

The following example repeatedly calls `read_str` to generate an array of strings.

```
// read_aos(len) reads in all non-whitespace strings from I/O
// and returns an array of those strings
// notes: modifies *len to store the length of the array
// returns NULL if there are no strings
// effects: allocates memory
// (caller must free each string and the array itself)
// modifies *len
// reads input
// time: O(n) where n is the length of all strings (combined)

char **read_aos(int *len);
```

example: reading in an array of strings

```
char **read_aos(int *len) {
    char **aos = NULL;
    int maxlen = 0;
    *len = 0;
    while (1) {
        char *s = read_str();
        if (s == NULL) break;
        if (*len == maxlen) {
            if (maxlen == 0) maxlen = 1;
            maxlen *= 2;
            aos = realloc(aos, maxlen * sizeof(char *));
        }
        aos[*len] = s;
        *len += 1;
    }
    if (*len < maxlen) {
        aos = realloc(aos, *len * sizeof(char *));
    }
    return aos;
}
```

The `read_aos` function uses the array doubling strategy on the array of pointers.

Each element in the array (*e.g.*, `aos[i]`) is a pointer to a string.

To properly free the array of strings, each string must be freed in addition to the array itself.

```
int len = 0;
char **aos = read_aos(&len);

// ...

// free all the strings and the aos itself
for (int i = 0; i < len; ++i) {
    free(aos[i]);
}
free(aos);
```

ADTs in C

With dynamic memory, we now have the ability to implement an *Abstract Data Type (ADT)* in C.

In Section 06, the first ADT we saw was a simple *stopwatch ADT*. It demonstrated **information hiding**, which provides both *security* and *flexibility*.

It used an **opaque** structure, which meant that the client could not **create** a stopwatch.

example: stopwatch ADT

This is the **interface** we used in Section 06.

```
// stopwatch.h [INTERFACE]

struct stopwatch;

// stopwatch_create() creates a new stopwatch at time 0:00
// effects: allocates memory (client must call stopwatch_destroy)
struct stopwatch *stopwatch_create(void);

// stopwatch_destroy(sw) frees memory for sw
// effects: sw is no longer valid
void stopwatch_destroy(struct stopwatch *sw);
```

We can now *complete* our **implementation**.

```
// stopwatch.c [IMPLEMENTATION]

struct stopwatch {
    int seconds;
};
// requires: 0 <= seconds

struct stopwatch *stopwatch_create(void) {
    struct stopwatch *sw = malloc(sizeof(struct stopwatch));
    sw->seconds = 0;
    return sw;
}

void stopwatch_destroy(struct stopwatch *sw) {
    assert(sw);
    free(sw);
}
```

Implementing a Stack ADT

As discussed in Section 06, the stopwatch ADT illustrates the principles of an ADT, but it is not a typical ADT.

The **Stack ADT** (one of the *Collection ADTs*) is more representative.

The interface is nearly identical to the stack implementation from Section 07 that demonstrated *oversized arrays*.

The only differences are: it uses an opaque structure, it provides create and destroy functions, and there is no maximum: it can store an arbitrary number of integers.

```
// stack.h (INTERFACE)

struct stack;

struct stack *stack_create(void);

bool stack_is_empty(const struct stack *s);

int stack_top(const struct stack *s);

int stack_pop(struct stack *s);

void stack_push(int item, struct stack *s);

void stack_destroy(struct stack *s);
```

The Stack ADT uses the “doubling” strategy.

```
// stack.c (IMPLEMENTATION)
```

```
struct stack {  
    int len;  
    int maxlen;  
    int *data;  
};
```

```
struct stack *stack_create(void) {  
    struct stack *s = malloc(sizeof(struct stack));  
    s->len = 0;  
    s->maxlen = 1;  
    s->data = malloc(s->maxlen * sizeof(int));  
    return s;  
}
```

```
void stack_destroy(struct stack *s) {  
    free(s->data);  
    free(s);  
}
```

Most of the operations are identical to the oversized array implementation.

```
bool stack_is_empty(const struct stack *s) {  
    assert(s);  
    return s->len == 0;  
}
```

```
int stack_top(const struct stack *s) {  
    assert(s);  
    assert(s->len);  
    return s->data[s->len - 1];  
}
```

```
int stack_pop(struct stack *s) {  
    assert(s);  
    assert(s->len);  
    s->len -= 1;  
    return s->data[s->len];  
}
```

The doubling strategy is implemented in push.

```
void stack_push(int item, struct stack *s) {
    assert(s);
    if (s->len == s->maxlen) {
        s->maxlen *= 2;
        s->data = realloc(s->data, s->maxlen * sizeof(int));
    }
    s->data[s->len] = item;
    s->len += 1;
}
```

What is the running time of a single call to `stack_push`?

- $O(n)$ when doubling occurs
- $O(1)$ otherwise (most of the time)

Amortized analysis

To understand *amortized analysis*, we first consider a more abstract example than `stack_push`.

Homer wants to do some “push-ups” to get some exercise.

His strategy is that on day k , when k is a power of 2, he will do k push-ups. He will then skip $(k - 1)$ days until it is another power of 2.

So the number of push-ups Homer does on the first 31 days is:

1,2,0,4,0,0,0,8,0,0,0,0,0,0,0,16,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

After 31 days, he has done exactly 31 push-ups. So, **on average**, he is doing one push-up per day.

The analysis for `stack_push` is very similar.

Ignoring any pop operations, the **total time** for n calls to `stack_push` is $O(n)$.

The ***amortized*** (“average”) time for **each call** is:

$$O(n)/n = O(1).$$

In other words, we can say that the *amortized* running time of `stack_push` is $O(1)$.

```
// stack_push(item, s) pushes item onto stack s
// requires: s is a valid stack
// effects:  modifies s
// time:    0(1) [amortized]
```

You will use *amortized* analysis in CS 240 and in CS 341.

In this implementation, we never “*shrink*” the array when items are popped.

A popular strategy is to shrink when the length reaches $\frac{1}{4}$ of the maximum capacity. Although more complicated, this also has an *amortized* run-time of $O(1)$ for an arbitrary sequence of pushes and pops.

Languages that have a built-in resizable array (e.g., C++’s vector) often use a similar “doubling” strategy.

Goals of this Section

At the end of this section, you should be able to:

- describe the heap
- use the functions `malloc`, `realloc` and `free` to interact with the heap
- explain that the heap is finite, and demonstrate how to check `malloc` for success
- describe memory leaks, how they occur, and how to prevent them

- describe the doubling strategy, and how it can be used to manage dynamic arrays to achieve an amortized $O(1)$ run-time for additions
- create dynamic resizable arrays in the heap
- write functions that create and return a new struct
- document dynamic memory side-effects in contracts