Dynamic Memory & ADTs in C

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 your program.

Memory is allocated from the heap upon request.
The heap

This memory is “borrowed”, and must 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.
The heap – Advantages

**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.
The heap: `malloc`

The `malloc` (memory allocation) function obtains memory from the heap dynamically. It is provided in `<stdlib.h>`.

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

For example, for an array of 10 `int`:

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

Or a single `struct posn`:

```c
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.
The heap: \texttt{malloc}

\begin{verbatim}
int *data = malloc(10 * sizeof(int));

struct posn *my_posn = malloc(sizeof(struct posn));
\end{verbatim}
The heap: Initialization

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

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

<table>
<thead>
<tr>
<th></th>
<th>0x80</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x84</td>
<td>???</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>my_posn</td>
<td>0xEE0</td>
<td>0x80</td>
</tr>
</tbody>
</table>
```
The heap: Traversal

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

```c
int *my_array = malloc(n * sizeof(int));
// array-index notation
for (int i = 0; i < n; ++i) {
    my_array[i] = 0;
}
// pointer notation
for (int *p = my_array; p < my_array + n; ++p) {
    *p = 0;
}
```
The heap: Size

You should always use `sizeof` with `malloc` to improve portability and to improve communication.

Seashell allows

```c
int *my_array = malloc(40);
```

instead of

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

but the latter is much better style and is more portable.
The heap: calloc

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.
The heap: malloc

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. Seashell is mostly forgiving, but in other C environments using an int when C expects a size_t may generate a warning. The proper printf placeholder to print a size_t is "%zd".
The heap: `malloc`

The declaration for the `malloc` function is:

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

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

```c
int *my_array = malloc(10 * sizeof(int));
struct posn *my_posn = malloc(sizeof(struct posn));
```
The heap: Visualization

```c
int main(void) {
    int *arr1 = malloc(5 * sizeof(int));
    int *arr2 = malloc(3 * sizeof(int));
    //...
}
```

<table>
<thead>
<tr>
<th>Void *</th>
<th>0x50</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x54</td>
<td>???</td>
<td></td>
</tr>
<tr>
<td>0x58</td>
<td>???</td>
<td></td>
</tr>
<tr>
<td>0x5C</td>
<td>???</td>
<td></td>
</tr>
<tr>
<td>0x60</td>
<td>???</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Void *</td>
<td>0x70</td>
<td>???</td>
</tr>
<tr>
<td>0x74</td>
<td>???</td>
<td></td>
</tr>
<tr>
<td>0x78</td>
<td>???</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arr1</th>
<th>Int *</th>
<th>0xE0</th>
<th>0x50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arr2</td>
<td>Int *</td>
<td>0xE8</td>
<td>0x70</td>
</tr>
</tbody>
</table>
The heap: Out of memory

An unsuccessful call to \texttt{malloc} returns \texttt{NULL}.

In practice, it’s good style to check every \texttt{malloc} return value and gracefully handle a \texttt{NULL} instead of crashing.

```c
int *my_array = malloc(n \times sizeof(int));
if (my_array == NULL) {
    printf("Sorry, out of memory! Exiting ...\n");
    exit(EXIT_FAILURE);
}
```

In the “real world” you should always perform this check, but in this course, you do not have to check for a \texttt{NULL} return value unless instructed otherwise.
The heap: **free**

For every block of memory obtained through `malloc`, you must eventually free the memory (when the memory is no longer in use).

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

In the Seashell environment, you must free every block.

```c
int *my_array = malloc(n * sizeof(int));
// ...
free(my_array);
```
The heap: `free`

Once a block of memory is freed, reading from or writing to it is invalid and may cause errors (or unpredictable results):

```c
int *data = malloc(5 * sizeof(int));
free(data);
int nope = data[0]; // Error: heap-use-after-free
```

The memory data is pointing at has been freed and is now invalid, but data is still pointing at it. `free(data)` does not modify the pointer data.
The heap: `free`

In Seashell, a runtime-error occurs if heap-memory is freed more than once:

```c
int *data = malloc(5 * sizeof(int));
free(data);
// ...
free(data);  // Error: double-free
```

Rule-of-thumb: for each call to `malloc` (and most calls to `realloc`), there must be a call to `free`. 
The heap: `free`

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

```c
int *data = malloc(5 * sizeof(int));
free(data); // Invalidates memory
data = NULL; // Good style
free(data); // free(NULL) is safe: it does nothing
```

<table>
<thead>
<tr>
<th>void *</th>
<th>0x50</th>
<th>INVALID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x54</td>
<td>INVALID</td>
<td></td>
</tr>
<tr>
<td>0x58</td>
<td>INVALID</td>
<td></td>
</tr>
<tr>
<td>0x5C</td>
<td>INVALID</td>
<td></td>
</tr>
<tr>
<td>0x60</td>
<td>INVALID</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>data</th>
<th>int *</th>
<th>0xE0</th>
<th>0x00</th>
</tr>
</thead>
</table>
The heap: \texttt{free}

In Seashell, a runtime-error occurs if memory that was not returned by a \texttt{malloc} is freed:

\begin{verbatim}
int data[5] = {0}; // Stack array
free(data); // ERROR: free-non-malloced-address
\end{verbatim}

Remember: you only have to manage \texttt{heap-memory} yourself; \texttt{stack-memory} is handled by the runtime environment.

\begin{verbatim}
int *data = malloc(5 * sizeof(int)); // Heap array
free(data + 1); // ERROR: free-non-malloced-address
\end{verbatim}
The heap: Memory leaks

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

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

```c
int *data;
data = malloc(5 * sizeof(int));
data = malloc(3 * sizeof(int)); // Memory leak!
free(data);
```

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.
The heap: Memory leaks

```c
int *arr;
arr = malloc(5 * sizeof(int));
arr = malloc(3 * sizeof(int)); // Memory leak!
```

![Heap diagram showing memory allocation and leak]
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.
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 Sort

// 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: O(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 arr[], int arr_len) {
    if (arr_len > 1) { // recursive condition
        int len_left = arr_len / 2;  // left half of arr
        int len_right = arr_len - len_left; // right half of arr
        int *arr_left = malloc(len_left * sizeof(int));
        int *arr_right = malloc(len_right * sizeof(int));
        for (int i = 0; i < len_left; ++i) arr_left[i] = a[i];
        for (int i = 0; i < len_right; ++i) arr_right[i] = a[i + len_left];
        merge_sort(arr_left, len_left); // recursively sort left sub-array
        merge_sort(arr_right, len_right); // recursively sort right sub-array
        merge(arr, arr_left, len_left, arr_right, len_right);
        free(arr_left);
        free(arr_right);
    }
}

Merge sort is $O(n \log n)$, even in the worst case.
The heap: Scope & side effects

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

```c
// build_array(n) returns a new array initialized with values a[0]=0, a[1]=1, ..., a[n-1]=n-1
// requires: len > 0
// effects: allocates heap memory (caller must free)
// time: O(n)
int *build_array(int len) {
    assert(len > 0);
    int *arr = malloc(len * sizeof(int));
    for (int i = 0; i < len; ++i) {
        arr[i] = i;
    }
    return arr;  // arr exists beyond function return
}
```
The heap: Scope & side effects

```c
int *build_array(int len) {
    assert(len > 0); // (1)
    int *arr = malloc(len * sizeof(int)); // (2)
    for (int i = 0; i < len; ++i) {
        arr[i] = i;
    }
    return arr; // (3)
}

int main(void) {
    int *my_array = build_array(3); // (4)
    free(my_array); // (5)
}
```

```
| 0x60 | ??? |
| 0x61 | ??? |
| 0x62 | ??? |
| ...  | ... |
| arr  | int * 0xE0 | ??? |
| ...  | ...      |
| my_array | int * 0xF0 | ??? |

void * 0x60 | ??? | 1 | 1 | INV

| 0x61 | ??? | 2 | 2 | INV
| 0x62 | ??? | 3 | 3 | INV

0x60 0x60 ...
0x60 ...
```

The diagram represents the memory allocation and usage of variables in the `build_array` function and the main function. The heap allocation and deallocation are shown with `malloc` and `free` functions respectively.
The heap: Scope & side effects

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):

```c
// build_array(n) returns a new array initialized
//   with values a[0]=0, a[1]=1, ..., a[n-1]=n-1
// requires: len > 0
// effects: allocates heap memory (caller must free)
// time: O(n)
int *build_array(int len);
```
A function could also free memory it did not allocate. That would also be a side effect:

```c
// process_array(arr, arr_len) ...
// requires: arr is heap-allocated
// effects: frees arr (arr is now invalid)
void process_array(int * arr, int arr_len) {
    // ...
    free(arr)
    // ...
}
```

This behaviour is rare outside of ADTs.
strdup

The `<string.h>` function `strdup` makes a duplicate of a string.

```c
// my_strdup(str) makes a duplicate of str.
// effects: allocates heap memory (caller must free)
char *my_strdup(const char *str) {
    char *new_str = malloc((strlen(str) + 1) * sizeof(char));
    strcpy(new_str, str);
    return new_str;
}
```

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, do not 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
Resizing arrays

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));

//...

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

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

```c
// realloc(ptr, newsize) resizes the memory block at
// ptr to be newsize and returns a pointer to the
// new location, or NULL if unsuccessful
// requires: ptr is heap-allocated
// effects: re-allocates heap memory (ptr is now
//          invalid, caller must free)
// time: 0(n), where n is min(newsize, oldsize)
```

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

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

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.
realloc

Be careful using realloc inside of a helper function.

```c
// repeat(str) modifies str by repeating it ("abc" => "abcabc")
//   and returns the new string.
// requires: str is a heap-allocated string
// effects: re-allocates memory (str is now invalid, caller must
//          free)
char *repeat(char *str) {
    int str_len = strlen(str);
    str = realloc(str, (str_len * 2 + 1) * sizeof(char));
    for (int i = 0; i < str_len; ++i) {
        str[i + str_len] = s[i];
    }
    str[str_len * 2] = '\0';
    return str; // this is essential!
}
```

A common mistake is to make repeat a void function (not return the new address for str). This causes a memory leak if the address of str changes.
realloc

Although rare, in practice,

```c
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.

```c
// 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.

```c
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.
String I/O: strings of unknown length

// read_str() 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(void) {
    char input;
    if (scanf("%c", &input) != 1) return NULL; // ignore initial whitespace
    int len_so_far = 1;
    char *str = malloc(len_so_far * sizeof(char));
    *str = input;
    while (true) { // loop until ...
        if (scanf("%c", &input) != 1) break; // EOF encountered or
        if (input == ' ' || input == '\n') break; // whitespace encountered
        ++len_so_far;
        str = realloc(str, len_so_far * sizeof(char)); // realloc one more byte
        str[len_so_far - 1] = input; // write input char
    }
    str = realloc(str, (len_so_far + 1) * sizeof(char)); // make space for '\0'
    str[len_so_far] = '\0';
    return str;
}
String I/O: strings of unknown length

Unfortunately, the running time of \texttt{read\_str} is $O(n^2)$, where $n$ is the length of the string. This is because \texttt{realloc} is $O(n)$ and occurs inside of the loop.

A better approach might be to allocate more memory than necessary and only call \texttt{realloc} when the array is \textquote{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 \textquote{actual} length in addition to the allocated length.
String I/O: strings of unknown length

// time: O(n) [see analysis on next slide]
char *read_str(void) {
    char input;
    if (scanf("%c", &input) != 1) return NULL; // ignore initial whitespace
    int len_so_far = 1;
    int len_max = 1;
    char *str = malloc(len_so_far * sizeof(char));
    *str = input;
    while (true) { // loop until ...
        if (scanf("%c", &input) != 1) break; // EOF encountered or
        if (input == ' ' || input == '\n') break; // whitespace encountered
        if (len == maxlen) {
            len_max *= 2; // doubling of str, if necessary
            str = realloc(str, len_max * sizeof(char));
        }
        ++len_so_far;
        str[len_so_far - 1] = input; // write input char
    }
    str = realloc(str, (len_so_far + 1) * sizeof(char)); // shrink to actual size
    str[len_so_far] = '\0';
    return str;
}
String I/O: strings of unknown length

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:

2, 4, 0, 8, 0, 0, 0, 16, 0, 0, 0, 0, 0, 32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 64

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

$$2 + 4 + 8 + \cdots + \frac{n}{4} + \frac{n}{2} + n + 2n = 4n - 2 = O(n)$$

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

To understand amortized analysis, let’s explore, when the str has to be resized.

| len_so_far | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| len_max    | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 1 | 6 | 1 | 6 | 1 | 6 | 1 | 6 | 1 | 6 | 1 | 6 | 1 | 6 | 1 | 6 | 1 | 6 | 1 | 6 | 1 |
| realloc    | ✓ | ✓ | ✓ | ✓ | ✓ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | ✓ |
| T(n)       | 1 | 2 | 4 | 8 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | 1 | 6 |   |   |   |   |   |   | 3 | 2 |

After reading 16 characters, runtime of realloc is $T(1) + T(2) + T(4) + T(8) = T(15)$.

After reading 17 characters, runtime of realloc is $T(1) + T(2) + T(4) + T(8) + T(16) = T(31)$.

After reading 32 characters, runtime of realloc is $T(1) + T(2) + T(4) + T(8) + T(16) = T(31)$.

So, on average, runtime is $\frac{T(n)}{n}$.
Amortized analysis

<table>
<thead>
<tr>
<th>len_so_far</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>len_max</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>realloc</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>T(n)</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

The image contains a table and a graph illustrating the amortized analysis for an algorithm. The table shows the values of `len_so_far`, `len_max`, and `realloc` across different iterations, while the graph visualizes the trend of `T(n)` over time.
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:

```c
// 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 heap memory (caller must free the
// array and each string)
// modifies *len
// reads input
// time: O(n) where n is the combined length of all strings
char **read_aos(int *len);
```
Reading in an array of strings

```c
char **read_aos(int *len_cur) {
    char **aos = NULL;
    int len_max = 0;
    *len_cur = 0;
    while (true) {
        char *str = read_str();
        if (str == NULL) break;
        if (*len_cur == len_max) {
            len_max = (len_max == 0) ? 1 : len_max * 2;
            aos = realloc(aos, len_max * sizeof(char *));
        }
        aos[*len_cur] = str;
        *len_cur += 1;
    }
    if (*len_cur < len_max) {
        aos = realloc(aos, *len_cur * sizeof(char *));
    }
    return aos;
}
```
Reading in an array of strings

```c
char **read_aos(int *len_cur) {
    int len_max = 0;
    *len_cur = 0;
    char **aos = NULL; // <- here
    while (true) {
        char *str = read_str();
        if (str == NULL) break;
        if (*len_cur == len_max) {
            len_max = (len_max == 0) ? 1 : len_max * 2;
            aos = realloc(aos, len_max * sizeof(char *));
        }
        aos[*len_cur] = str;
        *len_cur += 1;
    }
    if (*len_cur < len_max) {
        aos = realloc(aos, *len_cur * sizeof(char *));
    }
    return aos;
}

int main(void) {
    int aos_len = 9001;
    char **my_aos = read_aos(&aos_len);
    // ...
}
```
Reading in an array of strings

```c
char **read_aos(int *len_cur) {
    int len_max = 0;
    *len_cur = 0;
    char **aos = NULL;
    while (true) {
        char *str = read_str(); // <- here
        if (str == NULL) break;
        if (*len_cur == len_max) {
            len_max = (len_max == 0) ? 1 : len_max * 2;
            aos = realloc(aos, len_max * sizeof(char *));
        }
        aos[*len_cur] = str;
        *len_cur += 1;
    }
    if (*len_cur < len_max) {
        aos = realloc(aos, *len_cur * sizeof(char *));
    }
    return aos;
}

int main(void) {
    int aos_len = 9001;
    char **my_aos = read_aos(&aos_len);
    // ...
}
```
Reading in an array of strings

```c
char **read_aos(int *len_cur) {
    int len_max = 0;
    *len_cur = 0;
    char **aos = NULL;
    while (true) {
        char *str = read_str();
        if (str == NULL) break;
        if (*len_cur == len_max) {
            len_max = (len_max == 0) ? 1 : len_max * 2;
            aos = realloc(aos, len_max * sizeof(char *)); // <- here
        }
        aos[*len_cur] = str;
        *len_cur += 1;
    }
    if (*len_cur < len_max) {
        aos = realloc(aos, *len_cur * sizeof(char *));
    }
    return aos;
}

int main(void) {
    int aos_len = 9001;
    char **my_aos = read_aos(&aos_len);
    // ...
}
```
Reading in an array of strings

```c
char **read_aos(int *len_cur) {
    int len_max = 0;
    *len_cur = 0;
    char **aos = NULL;
    while (true) {
        char *str = read_str();
        if (str == NULL) break;
        if (*len_cur == len_max) {
            len_max = (len_max == 0) ? 1 : len_max * 2;
            aos = realloc(aos, len_max * sizeof(char *));
        }
        aos[*len_cur] = str;
        *len_cur += 1; // <- here
    }
    if (*len_cur < len_max) {
        aos = realloc(aos, *len_cur * sizeof(char *));
    }
    return aos;
}

int main(void) {
    int aos_len = 9001;
    char **my_aos = read_aos(&aos_len);
    // ...
}
```
Reading in an array of strings

```c
char **read_aos(int *len_cur) {
    int len_max = 0;
    *len_cur = 0;
    char **aos = NULL;
    while (true) {
        char *str = read_str();
        if (str == NULL) break;
        if (*len_cur == len_max) {
            len_max = (len_max == 0) ? 1 : len_max * 2;
            aos = realloc(aos, len_max * sizeof(char *));
        }
        aos[*len_cur] = str;
        *len_cur += 1;
    }
    if (*len_cur < len_max) { // <- here
        aos = realloc(aos, *len_cur * sizeof(char *));
    }
    return aos;
}

int main(void) {
    int aos_len = 9001;
    char **my_aos = read_aos(&aos_len);
    // ...
}
```
Reading in an array of strings

```c
char **read_aos(int *len_cur) {
    int len_max = 0;
    *len_cur = 0;
    char **aos = NULL;
    while (true) {
        char *str = read_str();
        if (str == NULL) break;
        if (*len_cur == len_max) {
            len_max = (len_max == 0) ? 1 : len_max * 2;
            aos = realloc(aos, len_max * sizeof(char *));
        }
        aos[*len_cur] = str;
        *len_cur += 1;
    }
    if (*len_cur < len_max) {
        aos = realloc(aos, *len_cur * sizeof(char *));
    }
    return aos;
}

int main(void) {
    int aos_len = 9001;
    char **my_aos = read_aos(&aos_len); // <- here
    // ...
    return 0;
}
```
Reading in an array of strings – aos as return value

```c
char **read_aos(int *len_cur) {
    int len_max = 0;
    *len_cur = 0;
    char **aos = NULL;
    while (true) {
        char *str = read_str();
        if (str == NULL) break;
        if (*len_cur == len_max) {
            len_max = (len_max == 0) ? 1 : len_max * 2;
            aos = realloc(aos, len_max * sizeof(char *));
        }
        aos[*len_cur] = str;
        *len_cur += 1;
    }
    if (*len_cur < len_max) { // <- here
        aos = realloc(aos, *len_cur * sizeof(char *));
    }
    return aos;
}

int main(void) {
    int aos_len = 9001;
    char **my_aos = read_aos(&aos_len);
    // ...
}
```

Reading in an array of strings – len as return value

```c
int read_aos(char ***aos) {
    int len_max = 0;
    int len_cur = 0;
    *aos = NULL;
    while (true) {
        char *str = read_str();
        if (str == NULL) break;
        if (len_cur == len_max) {
            len_max = (len_max == 0) ? 1 : len_max * 2;
            *aos = realloc(*aos, len_max * sizeof(char *));
        }
        (*aos)[len_cur] = str;
        len_cur += 1;
    }
    if (len_cur < len_max) { // <- here
        *aos = realloc(*aos, len_cur * sizeof(char *));
    }
    return len_cur;
}

int main(void) {
    char **my_aos;
    int aos_len = read_aos(&my_aos);
    // free everything
}
```
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.
ADTs in C: stopwatch

This is the interface we used in Section 06.

```c
// stopwatch.h [INTERFACE]
struct stopwatch;

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

// sw_destroy(sw) frees memory for sw
// effects: sw is no longer valid
void sw_destroy(struct stopwatch *sw);
```
ADTs in C: stopwatch

// stopwatch.c [IMPLEMENTATION]

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

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

void sw_destroy(struct stopwatch *sw) {
    assert(sw);
    free(sw);
}
ADTs in C: stack

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 maximum-length 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.
ADTs in C: stack

// stack.h [INTERFACE]

struct stack;
struct stack *stack_create(void);
bool stack_is_empty(const struct stack *stck);
int stack_top(const struct stack *stck);
int stack_pop(struct stack * stck);
void stack_push(struct stack *stck, int item);
void stack_destroy(struct stack *stck);
ADTs in C: stack

// stack.c [IMPLEMENTATION]

struct stack {
    int len_cur;
    int len_max;
    int *data;
};
ADTs in C: stack

// stack.c [IMPLEMENTATION]

struct stack {
    int len_cur;
    int len_max;
    int *data;
};

struct stack *stack_create(void) {
    struct stack *stck = malloc(sizeof(struct stack));
    stck->len_cur = 0;
    stck->len_max = 1;
    stck->data = malloc(stck->len_max * sizeof(int));
    return stck;
}
ADTs in C: stack

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

<table>
<thead>
<tr>
<th>void *</th>
<th>0x50</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x54</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0x58</td>
<td>0x70</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>void *</th>
<th>0x70</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x58</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0x58</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>0x58</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>strck</th>
<th>struck stack *</th>
<th>0xE0</th>
<th>0x50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0xE0</td>
<td>0x50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xE0</td>
<td>0x50</td>
</tr>
</tbody>
</table>
ADTs in C: stack

Most of the operations are identical to the maximum-length implementation.

```c
bool stack_is_empty(const struct stack *stck) {
    assert(stck);
    return stck->len_cur == 0;
}
```
ADTs in C: stack

```c
int stack_top(const struct stack *stck) {
    assert(stck);
    assert(stck->len_cur); // stck->len_cur != 0
    return stck->data[stck->len_cur - 1];
    // *(stck->data + stck->len_cur - 1)
}

int stack_pop(const struct stack *stck) {
    assert(stck);
    assert(stck->len_cur); // stck->len_cur != 0
    stck->len_cur -= 1;
    return stck->data[stck->len_cur];
    // *(stck->data + stck->len_cur)
}
```

<table>
<thead>
<tr>
<th>void *</th>
<th>0x50</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x54</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0x58</td>
<td>0x70</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>void *</th>
<th>0x70</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x58</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0x58</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>0x58</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

| stck | struck stack * | 0xE0 | 0x50 |
ADTs in C: stack

The doubling strategy is implemented in push.

```c
void stack_push(struct stack *stck, int item) {
    assert(stck);
    if (stck->len_cur == stck->len_max) {
        stck->len_max *= 2;
        stck->data = realloc(stck->data, stck->len_max * sizeof(int));
    }
    stck->data[stck->len_cur] = item;
    stck->len_cur += 1;
}
```

ADTs in C: stack

The doubling strategy is implemented in push.

```c
t void stack_push(struct stack *stck, int item) {
    assert(stck);
    if (stck->len_cur == stck->len_max) {
        stck->len_max *= 2;
        stck->data = realloc(stck->data, stck->len_max * sizeof(int));
    }
    stck->data[stck->len_cur] = item; // *(stck->data + stck->len_cur)
    stck->len_cur += 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, let’s explore, when the stack has to be resized.

| len_cur | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| len_max | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 1 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 3 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 6 |
| realloc | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| T(n)    | 1 | 2 | 4 | 8 | 8 | 8 | 1 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 3 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

After 16 stack_push, runtime of realloc is $T(1) + T(2) + T(4) + T(8) = T(15)$.

After 32 stack_push, runtime of realloc is $T(1) + T(2) + T(4) + T(8) + T(16) = T(31)$.

So, on average (i.e., per stack_push), runtime is $T(n)/n$. 
# Amortized analysis

<table>
<thead>
<tr>
<th>len_cur</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(n)</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>len_max</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>realloc</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T(n)</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing len_so_far, T(n), T(n) (acc)](image-url)
Amortized analysis

Ignoring any pop operations, the total time for \( n \) calls to \texttt{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 \texttt{stack\_push} is \( O(1) \).

// \texttt{stack\_push(item, stck)} pushes item onto stack \texttt{stck}
// requires: \texttt{stck} is a valid stack
// effects: modifies \texttt{stck}
// time: \( O(1) \) [amortized]
Amortized analysis

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