Dynamic Memory & ADTs in C

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

The heap

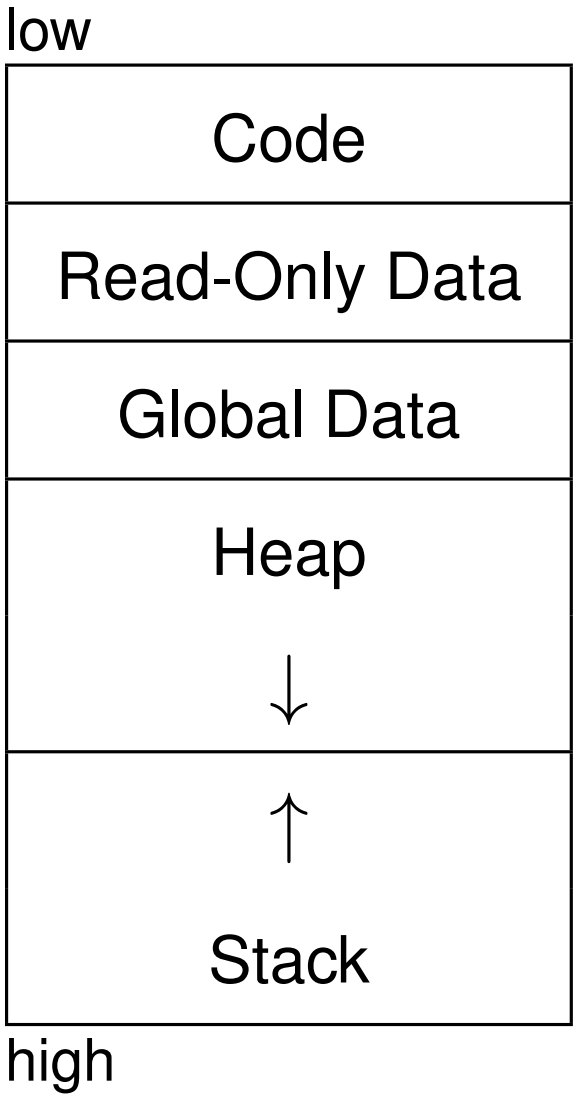
The ***heap*** is the final section in the C memory model.

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

Memory is **dynamically** *“borrowed”* from the heap. We call this ***allocation***.

When the borrowed memory is no longer needed, it can be *“returned”* and possibly **reused**. We call this ***deallocation***.

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


low
Code
Read-Only Data
Global Data
Heap
Stack
high

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 **alloc**ation) function obtains memory from the heap *dynamically*. It is provided in <`stdlib.h`>.

```
// malloc(s) requests s bytes of memory from the heap
//   and 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, if you want enough space for an array of 100 `int`s:

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

or an array of n `struct posn`s:

```
struct posn *my_posn_array = malloc(n * sizeof(struct posn));
```

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

`Seashell` will allow

```
int *my_array = malloc(400);
```

instead of

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

but the latter is much better style and is more portable.

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 declaration for the `malloc` function is:

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

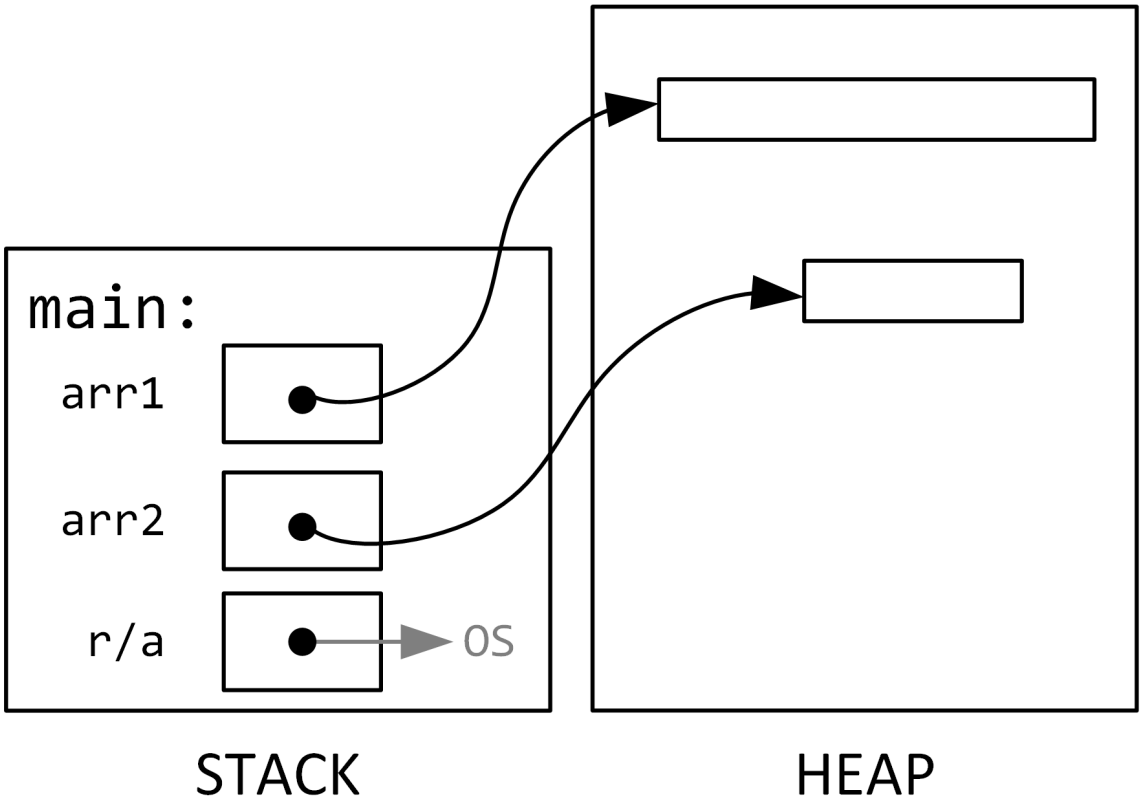
The return type is a (`void *`) (*void pointer*), a special pointer that can point at *any* type.

```
int *pi = malloc(sizeof(int));

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

example: visualizing the heap

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



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) {
  printf("Sorry dude, I'm out of memory! I'm 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 `NULL` return value unless instructed otherwise.

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

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

```
int *p = malloc(sizeof(int));
printf("the mystery value is: %d\n", *p);
```

Although `malloc` is very complicated, for the purposes of this course, you can assume that `malloc` is $O(1)$.

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.

free

For every block of memory obtained through `malloc`, you must eventually `free` the memory (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: O(1)
```

In the `Seashell` environment, you **must** `free` every block.

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

Invalid after free

Once a block of memory is `free`d, reading from or writing to that memory is invalid and may cause errors (or unpredictable results).

Similarly, it is invalid to `free` memory that was not returned by a `malloc` or that has already been `free`d.

```
int *p = malloc(sizeof(int));
free(p);
int k = *p;    // INVALID
*p = 42;       // INVALID
free(p);       // INVALID
p = NULL;      // GOOD STYLE
```

Pointer variables may still contain the address of the memory that was `free`d, so it is often good style to assign `NULL` to a `free`d pointer variable.

Memory leaks

A memory leak occurs when allocated memory is not eventually `free`d.

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

```
int *ptr;
ptr = malloc(sizeof(int));
ptr = malloc(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 `free`d.

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 Section 09 we saw a Racket implementation of the *divide and conquer* algorithm **merge sort** that is $O(n \log n)$.

In merge sort, the data is split into two smaller groups. After each smaller group is sorted, they are `merge`d together.

To simplify our C implementation, we will 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
// 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 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);
}
```

This implementation of merge sort is also $O(n \log n)$.

Duration

Using dynamic (heap) memory, a function can obtain memory that **persists after** the function has `return`ed.

```
// build_array(n) returns a new array initialized with
//   values a[0] = 0, a[1] = 1, ... a[n-1] = n-1
// effects: allocates a heap array (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
}
```

The caller (client) is responsible for `free`ing the memory (the contract should communicate this).

The <string.h> function strdup makes a duplicate 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 the include space for the null terminator.

strdup is not officially part of the C standard, but common.

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
- `free`ing 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));

// stuff happens...

// 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
// effects: the memory at p is invalid (freed)
// time: O(n), where n is newsize
```

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

```
int *my_array = malloc(100 * sizeof(int));
// stuff happens...
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**. You should assume that the parameter of `realloc` was `free`d and is now **invalid**.

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(NULL, s)` behaves the same as `malloc(s)`.

`realloc(ptr, 0)` behaves the same as `free(ptr)`.

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 `free`d.

```
// 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 size

In Section 08 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 `char`acter at a time.

```c
// readstr() reads in a new string from I/O
//   or returns NULL if EOF
// effects: allocates memory (caller must free)

char *readstr(void) {
  char c;
  // for the first char, ignore whitespace
  if (scanf(" %c", &c) != 1) return NULL;
  char *str = malloc(1 * sizeof(char));
  int len = 0;
  do {
    str[len] = c;
    ++len;
    str = realloc(str, (len + 1) * sizeof(char));
    if (scanf("%c", &c) != 1) break;
  } while (c != ' ' && c != '\n');
  str[len] = '\0';
  return str;
}
```

Amortized analysis

Unfortunately, the running time of `readstr` 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 size of the array when it is full.

Similar to working with *maximum-length arrays*, we need to keep track of the *"actual"* length in addition to the *allocated* length.

```c
char *readstr(void) {
  char c;
  if (scanf(" %c", &c) != 1) return NULL;
  int maxlen = 1;
  char *str = malloc(maxlen * sizeof(char));
  int len = 0;
  do {
    str[len] = c;
    ++len;
    if (len == maxlen) {    // DOUBLE the allocated array size
      maxlen *= 2;
      str = realloc(str, maxlen * sizeof(char));
    }
    if (scanf("%c", &c) != 1) break;
  } while (c != ' ' && c != '\n');
  str[len] = '\0';
  // shrink the array back down to the correct size
  str = realloc(str, (len + 1) * sizeof(char));
  return str;
}
```

With our "doubling" strategy, most iterations will be $O(1)$, unless it is necessary to resize (`realloc`) the array.

The resizing time for the first 32 iterations would be:

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

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

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

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

In other words, the ***amortized*** ("average") time for each iteration is:

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

ADTs in C

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

In Section 02, the first ADT we saw was a simple *account ADT*, which stored a username and a password. It demonstrated **information hiding**, which provides both *security* and *flexibility*.

We will also need to use **opaque** structures (incomplete declarations without fields), as introduced in Section 06.

example: account ADT

In the **interface**, we only provide an *incomplete declaration*. In addition to the normal operations, we provide functions to **create** and **destroy** instances of the ADT.

```
// account.h -- a simple account ADT module

struct account;                 // incomplete

// create_account(username, password) creates an account
//   with the given username and password
// effects: allocates memory (client must call destroy_account)
struct account *create_account(const char *username,
                               const char *password);

// destroy_account(acc) removes all memory for acc
// effects: memory at acc is free'd and invalid
void destroy_account(struct account *acc);
```

Because the interface only provides an incomplete declaration, the **client** does not know the fields of the `account` structure.

The client can only define a *pointer* to the structure, which is `returned` by `create_account`.

```
// client.c

char username[9];
char password[41];

// ...

struct account *my_account = create_account(username, password);

// ...

destroy_account(my_account);
```

The *complete* structure declaration only appears in the **implementation**.

```
// account.c

struct account {
  char *uname;
  char *pword;
};
```

`create_account` returns a *pointer* to a **new** account.

```
struct account *create_account(const char *username,
                               const char *password) {

  struct account *a = malloc(sizeof(struct account));

  a->uname = malloc((strlen(username) + 1) * sizeof(char));
  strcpy(a->uname, username);

  a->pword = malloc((strlen(password) + 1) * sizeof(char));
  strcpy(a->pword, password);

  return a;
}
```

it makes **duplicates** of the username and password strings provided by the client.

In C, our ADT also requires a `destroy_account` to `free` the memory created (both the fields and the structure itself).

```
void destroy_account(struct account *a) {
  free(a->username);
  free(a->password);
  free(a);
}
```

The remaining operations are straightforward.

```
const char *get_username(const struct account *acc) {
  return acc->uname;
}

bool is_correct_password(const struct account *acc,
                         const char *word) {
  return (strcmp(acc->pword, word) == 0);
}
```

Implementing a Stack ADT

As discussed in Section 02, the account 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 08 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.

```c
// stack.h (INTERFACE)

struct stack;

struct stack *create_stack(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(const struct stack *s);
```

The Stack ADT uses the “doubling” strategy. It is typical to have an initial size that is not too wasteful, but avoids excessive doubling for small stacks.

```
// stack.c (IMPLEMENTATION)

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

static const int initial_size = 32;

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

The doubling is implemented in `push`.

`destroy` must `free` the field and the structure itself.

```
// Time: O(1) [amortized]

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;
}

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

The remaining operations are identical to the maximum-length 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];
}
```

As discussed earlier, the *amortized* run-time for `push` is $O(1)$.

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 reduce the size 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 `push`es and `pop`s.

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 use 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