Dynamic Memory

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

Dynamic memory

When defining a C array, the length of the array must be known, but Racket lists can grow to be arbitrarily large without knowing in advance the length of the list.

Racket lists are `cons`tructed with **Dynamic memory**.

Dynamic memory is allocated from the **heap** *while the program is running* (more on this later).

The heap is the final section of memory in our memory model.

In Section 11 we use dynamic memory to construct Racket-like lists in C.

Maximum-length arrays

Before we introduce dynamic memory, we discuss a less sophisticated alternative.

In some applications, it may be "appropriate" to have a **maximum length** for an array. For example, in Section 08 we used `scanf` to store a name in an array with a maximum number of characters.

In general, maximums should only be used when appropriate. They can be very wasteful if the maximum is excessively large.

Many "real-world" systems have maximums. UW login ids are restricted to 8 characters and last names are restricted to `40`.

When working with maximum-length arrays, we need to keep track of

- the "**actual**" **length** of the array, and
- the **maximum possible length**.

```
const int numbers_max = 100;
int numbers[100];
int numbers_len = 0;

void append_number(int i) {
  assert(numbers_len < numbers_max);
  numbers[numbers_len] = i;
  numbers_len++;
}
```

This approach does not use dynamic memory, but the length of the array is "dynamic" (can change during run-time).

What if the maximum length is exceeded?

- An error message can be displayed.
- The program can `exit`.
- A special return value can be used.

```
// returns true if i was successfully appended,
// false otherwise
bool append_number(int i) {
  if (numbers_len == numbers_max) return false;
  numbers[numbers_len] = i;
  numbers_len++;
  return true;
}
```

Any approach may be appropriate as long as the contract properly documents the behaviour.

The `exit` function (part of <`stdlib.h`>) stops program execution.

It is useful for “fatal” errors.

```
#include <stdlib.h>

// effects: prints a message and exit upon failure
void append_number(int i) {
  if (numbers_len == numbers_max) {
    printf("FATAL ERROR: numbers_max exceeded\n");
    exit(EXIT_FAILURE);
  }
  //...
}
```

The `exit` argument is the same as the `main return` value.

For convenience, <`stdlib.h`> defines `EXIT_SUCCESS` (`0`) and `EXIT_FAILURE` (non-zero).

To make our maximum-length arrays more convenient and our code more re-usable, we can store the array information in a structure.

```
struct max_array {
  int *data;
  int len;
  int max;
};

void append_to_max_array(struct max_array *ma, int i) {
  if (ma->len < ma->max) {
    ma->data[ma->len] = i;
    ma->len++;
  } else { ... } // error
}

//example:
const int numbers_max = 100;
int numbers_data[100];
struct max_array numbers = {numbers_data, 0, numbers_max};
```

The heap

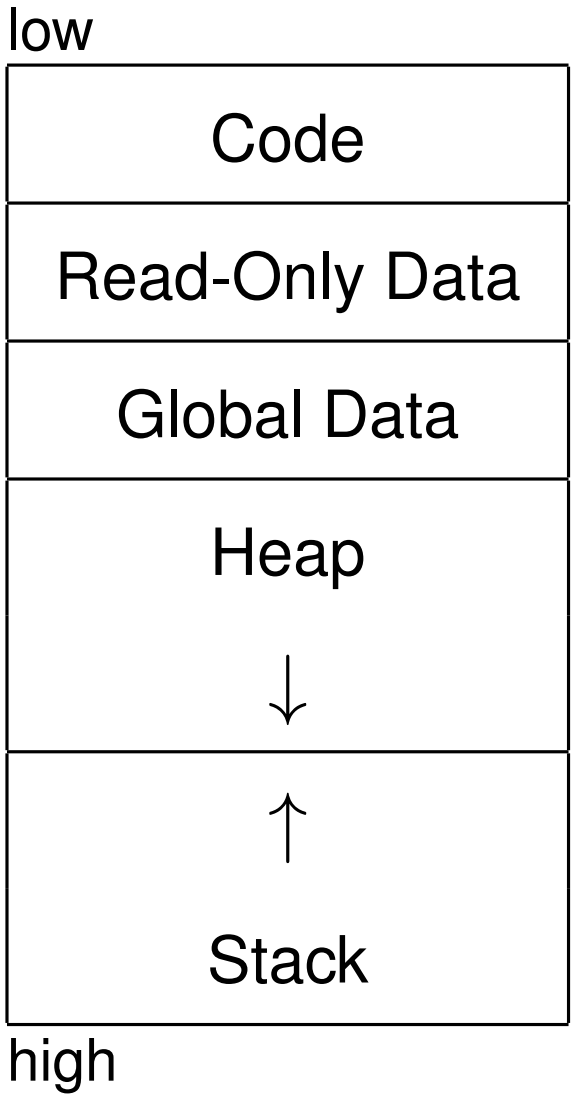
The ***heap*** is the final section in the C memory model.

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 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” stuck.

As we see later, there is also an *abstract data type* known as a stack, but because its behaviour is similar to “the stack”, its name is far less confusing.

malloc

The `malloc` (**m**emory **alloc**ation) function obtains memory from the heap dynamically (it is part of <`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
void *malloc(size_t s);
```

`size_t` is the type of integer produced by the `sizeof` operator. You should always use `sizeof` with malloc to improve portability and to improve communication.

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

Strictly speaking, `size_t` and `int` are different types.

`Seashell` allows `malloc(4)` instead of `malloc(sizeof(int))`, but the latter is much better style and is more portable.

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`.

`malloc` returns an address of type (`void *`) (*void pointer*) which can be assigned to a pointer variable of any type. Thus any type can be stored in the heap.

In this course, you should not run out of memory, but in practice it's good style to check every `malloc` return value and gracefully handle a `NULL` instead of crashing.

```
int *p = malloc(sizeof(int));
if (p == NULL) {
  printf("sorry dude, out of memory! I'm exiting.\n");
  exit(EXIT_FAILURE);
}
```

Unless otherwise noted, you do **not** have to check for a `NULL` return value in this course. The check is often omitted in these notes.

The memory on the heap returned 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

```
// free(p) returns memory at p back to the heap
// requires: p must be from a previous malloc
// effects:  the memory at p is no longer valid
void free(void *p);
```

For every block of memory obtained through `malloc`, you should eventually `free` the memory (when the memory is no longer in use). You can assume that `free` is $O(1)$.

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

Once a block of memory is `free`d, it cannot be accessed (and can not be `free`d a second time). It may be returned by a future `malloc` and become valid again.

memory sections

```c
const int r = 42;
int g = 15;
int main(void) {
  int s = 23;
  int *h = malloc(sizeof(int));
  *h = 16;
  printf("the address of main is:  %p\n", main); // CODE
  printf("the address of    r is:  %p\n", &r);   // READ-ONLY
  printf("the address of    g is:  %p\n", &g);   // GLOBAL DATA
  printf("the value   of    h is:  %p\n", h);    // HEAP
  printf("the address of    s is:  %p\n", &s);   // STACK
  free(h);
}
```

```
the address of main is:  0x46c060
the address of r is:     0x477700
the address of g is:     0x68a9e0
the value of h is:       0x60200000eff0
the address of s is:     0x7fff05ffc570
```

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 (it is lost) and so it can never be `free`d.

Invalid after free

Once a `free` occurs, one or more pointer variables may still contain the address of the memory that was `free`d.

Any attempt to read from or write to that memory is invalid, and can cause errors or unpredictable results.

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

It is often good style to assign `NULL` to a `free`d pointer variable to avoid misuse.

Garbage collection

In Racket, we are not concerned with `free`ing memory when it is no longer needed because Racket has a ***garbage collector***. Many modern languages have a garbage collector.

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

The biggest disadvantage of a garbage collector is that it can affect performance, which is a concern in high performance computing.

Many programmers believe the benefits of a garbage collector outweigh any disadvantages (many also believe that it is important to learn how to program without a garbage collector).

Dynamically allocating arrays

We can create a ***dynamic array*** in the heap.

```
// effects: allocates an array on heap (caller must free)
//          (description of error behaviour...)
int *create_heap_array(int len) {
  assert(len > 0);
  int *a = malloc(sizeof(int) * len);     // array size
  if (a == NULL) { ... }                  // can error check!
  return a;     // array can exist beyond function call
}
```

One of the key advantages of dynamic (heap) memory is that a function can "create" new memory that persists **after** the function has `return`ed.

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

The <string.h> function strdup duplicates a string by creating a new dynamically allocated array.

```
// my_strdup(s) makes a duplicate of s
//   returns NULL if there is not enough memory
// effects: allocates memory: caller must free
char *my_strdup(const char *s) {
  char *new = malloc(sizeof(char) * (strlen(s) + 1));
  strcpy(new,s);
  return new;
}
```

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

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

Resizing arrays

`malloc` is passed the size of the block of memory to be allocated.

On its own, this does not 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

Usually this is used to make a *larger* array, but if a smaller array is requested the extra elements are discarded.

```
// resize_array(old, oldlen, newlen) changes the length
//   of array old from oldlen to newlen
//   returns new array or NULL if out of memory
//   if larger, new elements are uninitialized
// requires: old must be a malloc'd array of size oldlen
// effects:  frees the old array, caller must free new array
// time: O(n), where n is min(oldlen,newlen)

int *resize_array(int *old, int oldlen, int newlen) {
  int *new = malloc(sizeof(int) * newlen);
  int copylen = oldlen;
  if (newlen < oldlen) copylen = newlen;
  for (int i=0; i < copylen; i++) {
    new[i] = old[i];
  }
  free(old);
  return new;
}
```

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

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

The `ptr` parameter in `realloc` must be a value returned from a **previous** `malloc` (or `realloc`) call.

Similar to our `resize_array`, `realloc` preserves the contents from the previous `malloc`, discarding the memory if `s` is smaller, and providing *uninitialized* memory if `s` is larger.

For this course, you can assume that `realloc` is $O(s)$.

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

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

`realloc` makes our `resize_array` more straightforward:

```
//  (oldlen is no longer required)
int *resize_array(int *old, int newlen) {
  return realloc(old, sizeof(int) * newlen);
}
```

The return pointer of `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**. It's common to overwrite the old pointer with the new value.

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

In practice,

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

could cause a memory leak.

In C99, if an "out of memory" condition occurs during `realloc`, the return value of `realloc` is `NULL` and `realloc` does not `free` the original memory block.

Amortized analysis

Consider the following code to dynamically increase an array every time a number is added:

```
int *numbers = NULL;
int numbers_len = 0;

void append_number(int i) {
  if (numbers_len == 0) {
    numbers = malloc(sizeof(int));
  } else {
    numbers = realloc(numbers, sizeof(int) * (numbers_len + 1));
  }
  numbers[numbers_len] = i;
  numbers_len++;
}
```

The time required to add n numbers is $\sum_{i=1}^{n} O(i) = O(n^2)$.

The previous `append_number` function called `realloc` *every* time a number was added.

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. We can use a structure to keep track of the lengths.

```
struct dyn_array {
  int *data;
  int len;
  int max;
};
```

```c
struct dyn_array {
  int *data;
  int len;
  int max;
};

void append_number(struct dyn_array *da, int i) {
  if (da->len == da->max) {
    if (da->max == 0) {
      da->max = 1;
      da->data = malloc(sizeof(int));
    } else {
      da->max *= 2;
      da->data = realloc(da->data, sizeof(int) * da->max);
    }
  }
  da->data[da->len] = i;
  da->len++;
}

// definition example
struct dyn_array numbers = {NULL, 0, 0};
```

With our "doubling" strategy, calls to `append_number` are $O(n)$ when resizing is necessary, and $O(1)$ otherwise.

Element-copying cost for first 33 calls to `append_number` would be:

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

For $n+1$ calls to `append_number`, the total copying cost is:

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

If we add 1 to each call for assigning the new value, the *total time* for n calls to `append_number` is

$$O(n) + O(n) = O(n).$$

Thus, the ***amortized*** ("averaged") run-time of `append_number` is

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

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

For arrays that can also *"shrink"* (with deletions), a popular strategy is to reduce the size when the length reaches $\frac{1}{4}$ of the capacity. Although more complicated, this also has an *amortized* run-time of $O(1)$ for an arbitrary sequence of inserts and deletions.

Some languages have a built-in resizable array (often called a `vector`) that uses a similar "doubling" strategy.

example: create & destroy a dynamic array

```
struct dyn_array *create_dyn_array(int initial_max) {
  struct dyn_array *da = malloc(sizeof(struct dyn_array));
  da->len = 0;
  da->max = initial_max;
  if (da->max > 0) {
    da->data = malloc(sizeof(int) * initial_max);
  } else {
    da->data = NULL;
  }
  return da;
}

void destroy_dyn_array(struct dyn_array *da) {
  if (da->max > 0) {
    free(da->data);
  }
  free(da);
}
```

Earlier we discussed how `struct` function parameters are typically "passed by pointer" to avoid copying the contents of the structure.

For a similar reason, functions also rarely `return` structures.

Typically, either:

- the function creates a **new** `struct` with `malloc` and returns a pointer (see the previous example),

  ```
  struct posn *make_me_a_new_posn(...) {...}
  ```

- or the calling function passes (a pointer to) an *existing* `struct` for the function to update (or initialize).

  ```
  void init_my_posn(struct posn *p, ...) {...}
  ```

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 the `exit` function if a program runs out of memory
- describe memory leaks, how they occur, and how to prevent them
- explain the advantages of creating an array in the heap instead of the stack

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