Dynamic Memory

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

Dynamic memory

When declaring a C array, the length of the array must be known.

However, Racket lists can grow to be arbitrarily large without knowing in advance how many elements will be in the list.

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

The memory requirements of a program are “dynamic” because they can change *while the program is running*.

In this unit we will introduce *dynamic memory* in C, and in the following unit we will use dynamic memory to construct Racket-like lists in C.

Maximum-length arrays

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

In some applications, it may be "appropriate" to have a **maximum length** for an array. For example, earlier we had a stored a name and there was a maximum number of characters available.

In general, maximums should only be used when necessary. 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[numbers_max];
int numbers_len = 0;

void add_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”.

What if the maximum length is exceeded?

- A special return value can be used.

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

- An error message can be displayed.
- The program can `exit`.

Any approach may be appropriate as long as the POST-conditions properly document the behaviour.

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

It is useful for “fatal” errors.

```
#include <stdlib.h>

// POST:  will print a message and exit upon failure
void add_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.

<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 add_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[numbers_max];
struct max_array numbers = {numbers_data, 0, numbers_max};
```

Heap

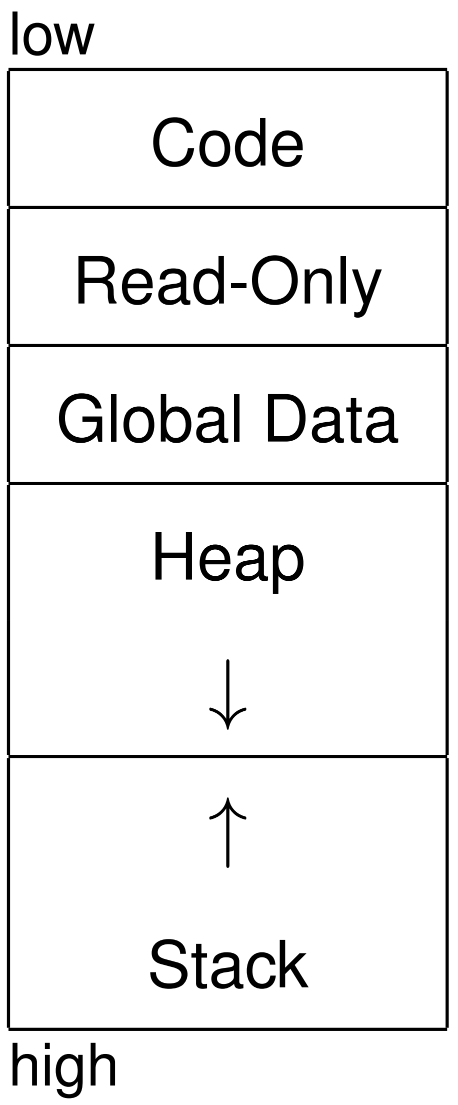
The ***heap*** is the final section in the C memory model.

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.

When the borrowed memory is no longer needed, it can be *“returned”* and reused.

Eventually, the heap will become empty, and any attempts to borrow additional memory will fail.


low
Code
Read-Only
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 will 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
//   POST: returns a pointer to a block of s bytes, or
//         NULL if not enough memory is available
void *malloc(size_t s);
```

The `sizeof` operator produces the integer size type (`size_t`). You should always use `sizeof` with malloc to improve portability and to communicate your intent.

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

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

In RunC, `size_t` and `int` are both 4 byte integers and are compatible. RunC will allow `malloc(4)` instead of `malloc(sizeof(int))`, but the latter is much better style.

In other C environments they may be different, and using an `int` when a C expects a `size_t` can generate warnings.

The proper `printf` placeholder to print a `size_t` is `%z`, although we use `%d` in the notes to avoid confusion.

```
// malloc(s) requests s bytes of memory from the heap
//   POST: returns a pointer to a block of s bytes, or
//         NULL if not enough memory is available
void *malloc(size_t s);
```

`malloc` returns a *void pointer* (`void *`), which can point at any type of memory and can be assigned to any pointer type.

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);
}
```

The memory on the heap returned by `malloc` is **uninitialized**.

```c
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 to the heap memory pointed to by p
//   PRE:  p must have been obtained by a previous malloc
//   POST: 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 needed). You can assume that `free` is $O(1)$.

In the RunC 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.

```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:   0x804d2e0
the address of    r is:   0x805ba60
the address of    g is:   0x8062080
the value   of    h is:  0xaff84f80
the address of    s is:  0xbfd432c0
```

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 (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.

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

A garbage collector will detect when memory is no longer needed and automatically return 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.

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

Dynamically allocating arrays

Previously, we saw how C99 arrays can be declared (on the stack) with a length determined *while the program is running*.

```
void uses_stack_array(int len) {
  assert(len > 0);
  int a[len];        // len determined at run-time
  ...
}
```

This may be “dynamic”, but it is not “dynamic memory” as it uses the stack, not the *heap*. This approach has three disadvantages:

- the array “disappears” when the function `return`s,
- the array could be large and overflow the stack, and
- the array cannot be *“resized”* (more on this later).

It is a much better strategy to create the array on the heap.

```
// POST: 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.

```
// POST: caller must free
//       returns NULL if there is not enough memory
char *my_strdup(const char *s) {
  char *new = malloc(sizeof(char) * (strlen(s) + 1));
  if (new) {
    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.

Resizing arrays

`malloc` requires 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 will be discarded.

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

int *resize_array(int *old, int oldlen, int newlen) {
  int *new = malloc(sizeof(int) * newlen);
  if (new) {
    int copylen = newlen < oldlen ? newlen : oldlen;
    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);
```

To avoid cluttering slide code, we will no longer show code to "handle" out of memory conditions. While important in practice, it is not a concern in this course.

```c
ptr = malloc(...); // or realloc(...)
if (ptr == NULL) {
  printf("FATAL ERROR: ran out of memory\n");
  exit(EXIT_FAILURE);
}
```

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 add_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 `add_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 will 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.

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

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

void add_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++;
}

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

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

The approx. run-time for the first 32 calls to `add_number` would be:
1,2,1,4,1,1,1,8,1,1,1,1,1,1,1,16,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,32

For n calls to `add_number`, the total time for the resizing calls is:
$n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 = 2n - 1 = O(n)$.

The total time for the non-resizing calls is also $O(n)$, so the *total time* for n calls to `add_number` is $O(n) + O(n) = O(n)$.

Therefore, the ***amortized*** ("averaged") run-time of `add_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.

C++ has a built-in resizable array called a `vector` that uses a similar “doubling” strategy.

For a final example of dynamic memory, the following code *dynamically* creates a new `struct dyn_array`.

```c
struct dyn_array *create_dyn_array() {
  struct dyn_array *a = malloc(sizeof(struct dyn_array));
  a->len = 0;
  a->max = 0;
  a->data = NULL;
  return a;
}
```

In practice, you may want to initialize `max` to a small but not wasteful size (*e.g.* 64) to avoid numerous initial calls to `realloc`.

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 module

You should understand that the heap provides dynamic memory and is the final section of the C memory model.

You should understand how to use the functions `malloc`, `realloc` and `free` to interact with the heap.

You should understand that the heap is finite, and how to use gracefully use the `exit` function if a program runs out of memory.

You should understand what a memory leak is, how they occur, and how to prevent them.

You should understand the advantages of creating an array in the heap instead of the stack.

You should understand how to create dynamic arrays in the heap, and how to resize them.

You should understand the maximum-length arrays strategy, and how a similar strategy can be used to manage dynamic arrays to achieve an amortized $O(1)$ run-time for additions.

You should understand how a function can create and return a new `struct`.

You should understand the importance of documenting dynamic memory side-effects in the contract.