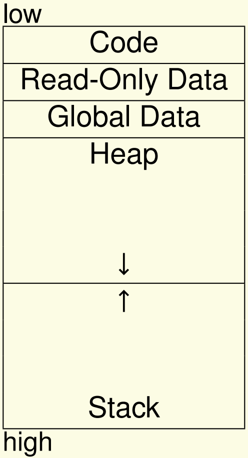
Module 9: Dynamic Memory & ADTs in C

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

It is like a big "pile" of memory that is available to your program.

There are just 2 things we can do with the heap:

1. We can "borrow" a block of memory from the heap, using the `malloc` function, for **m**emory **alloc**ation.
2. We can "give back" a block that we previously allocated, using the `free` function.

These are provided by `stdlib.h`, which `cs136.h` includes.

Memory from `malloc` is not in the stack. A stack frame can interact with the heap only via pointers.

“A heap” vs “the heap”

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 less confusing.

“A heap” vs “the heap”

`p = malloc(size)` reserves `size` bytes of memory; `p` points at it.

```
// make_str(ch, n) Return a new array containing a
// string containing n copies of ch.
// effects: allocates heap memory [caller must free]
char *make_str(char ch, int n) {
  char *p = malloc(n + 1);    // allocate n+1 chars.
  for (int i = 0; i < n; ++i) // write to them.
    p[i] = ch;
  p[n] = '\0'; // null-terminate the string
  // TRACE:1
  return p;
}
```

```
int main(void) {
  char *str = make_str('X', 4);
  // TRACE:2
  assert(0 == strcmp(str, "XXXX"));
}
```

Exercise: Draw memory diagrams at `TRACE:1` and `TRACE:2`.

Memory leaks

A memory leak occurs when allocated memory is not eventually freed.
Programs that leak memory may suffer degraded performance or eventually crash.
To avoid memory leaks, we must always:

- `free` memory that we allocate.
- For any function that allocates memory, and does not itself `free` it, **document** this effect clearly.

```
// effects: allocates heap memory [caller must free]
//          ^^^^^^^^^ ^^^^ ^^^^^^  ^^^^^^ ^^^^ ^^^^
char *make_str(char ch, int n) {
  char *p = malloc(n + 1);   // allocate n+1 chars.
  return p;
}
```

```
int main(void) {
  char *str = make_str('X', 4);
  assert(0 == strcmp(str, "XXXX"));
  free(str); // <-- important!
}
```

Use of malloc and free

The size of a single `char` is 1, so the size of `n char` values is `n`.
But normally, we should use `sizeof` to determine the size of a single item:

```
// make_squares(n) Return an array of
//    the squares of the first n nats.
// effects: allocates heap memory [caller must free]
int *make_squares(int n) {
  int *buf = malloc(n * sizeof(int)); // <--
  for (int i = 0; i < n; ++i) buf[i] = i*i;
  return buf;
}
```

```
int main(void) {
  int *sqs = make_squares(5);
  trace_array_int(sqs, 5);
  free(sqs);
}
```

```
>>> [main.c|main|24] >> sqs ⇒ [0, 1, 4, 9, 16]
```

Exercise

Create a function `int *fib_array(int n)` that creates a new array containing the first `n` Fibonacci numbers, for $n \geq 2$.
Call it several times with different values of `n`.

Use of `malloc` and `free`

To create an array of 100 `int` values, you should write:

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

It is *possible* to write:

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

This **assumes** that `sizeof(int)` ⇒ `4`. This is not a safe assumption!

- On Arduino and many other embedded systems, `sizeof(int)` ⇒ `2`.
- On certain Cray supercomputers, `sizeof(int)` ⇒ `8`.
- On SHARC DSPs, a `char` is 32 bits, and an `int` is the same size, so `sizeof(int)` ⇒ `1`.

Here "possible" means "C will not protect you from doing this unwise thing".

! Do not assume you know how big things are. Use `sizeof` every time.
(Except with `char`, since `sizeof(char)` is 1 by definition.)

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.

Our environment 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`.

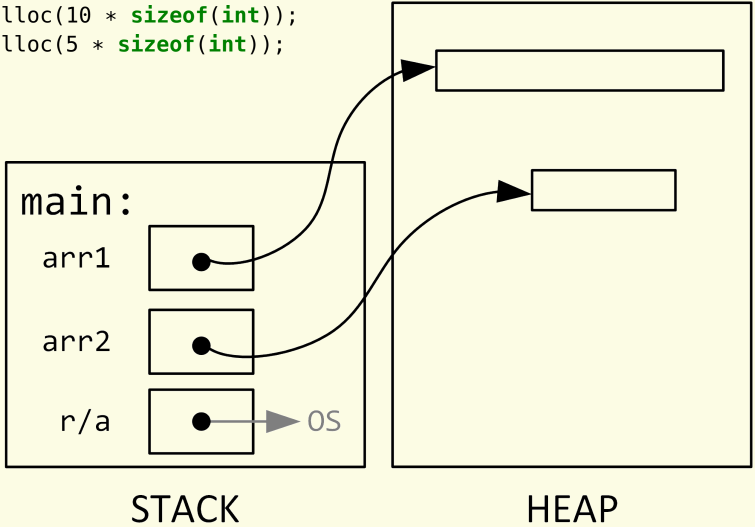
The declaration for the `malloc` function is:

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

The return type is a (`void *`) (*void pointer*).
This is 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));
```

```c
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, 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.

Working with `malloc`

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

```
int *a = malloc(10 * sizeof(int));
printf("the mystery value is: %d\n", a[0]);
```

For the purposes of this course, 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.

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

> **!** In our environment, you **must** `free` every block.
> The AddressSanitizer will (usually) point out the error if your program ends without freeing something.

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

`free` does not need to know the size of the memory block. C “remembers” the size, which is actually a bit surprising and magical given everything else we know about C.

Invalid after `free`

Once a block of memory is freed, reading from or writing to that memory is invalid.

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

```
int *a = malloc(10 * sizeof(int));
free(a);
int k = a[0];   // INVALID
a[0] = 42;      // INVALID
free(a);        // INVALID
a = NULL;       // GOOD STYLE
```

Pointer variables still contain the address of the memory that was freed.
It can be good style to assign `NULL` to a freed pointer variable.

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 into two separate arrays. The two arrays are sorted and then they are merged back together into the original array.

Exercise

Given the `merge` function, write

`void merge_sort(int *a, int len)`

that mutates `a` to be sorted.

```
// merge(dest, src0, len0, src1, len1)
//     Copy  values from src0 and src1 to dest,
//     so the result is sorted.
// requires: src0 and src1 are already sorted.
//           dest has size at least len0 + len1
void merge(int *dest, const int *src0, int len0,
                      const int *src1, int len1) {
  while (len0 || len1) {
    if (!len1 || (len0 && *src0 < *src1)) {
      *dest = *src0;
      len0--;
      src0++;
    } else {
      *dest = *src1;
      len1--;
      src1++;
    }
    dest++;
  }
}
```

An array created my `malloc` **persists**, that is, continues to exist, until we call `free`.

```
// make_squares(n) Return an array of
//    the squares of the first n nats.
// effects: allocates heap memory [caller must free]
int *make_squares(int n) {
  int *buf = malloc(n * sizeof(int)); // <--
  for (int i = 0; i < n; ++i) buf[i] = i*i;
  return buf;
}
```

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

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

Exercise

Write a function `char *my_strdup(const char *src)`.
It returns a pointer to heap memory that contains a copy of the string that was in `src`.

```
int main(void) {
  char word[] = "Not all those who wander are lost.";
  char *copy = my_strdup(word);
  assert(copy != word); // different addresses
  assert(0 == strcmp(copy, word)); // equal strings
  free(copy); // don't leak memory
}
```

Don't forget to include space for the null terminator!

`strdup` is not part of C99, but it is part of POSIX.1-2008.

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 **resize** an array, we can:

1. create a new array
2. copy the items from the old array to the new array
3. `free` the old array

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

// stuff happens...

// oops, arr now needs to have a length 101
int *old = arr;
arr = malloc(101 * sizeof(int));  // (1)
for (int i = 0; i < 100; ++i) {
  arr[i] = old[i];                // (2)
  }
free(old);                        // (3)
```

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 $\min(\texttt{newsize}, \texttt{oldsize})$
The cost comes from copying the values.

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 address passed to `realloc` was freed 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.

```
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
}
```

In Module 6 we saw how reading in strings can be susceptible to buffer overruns.

```
char str[81];
int retval = scanf("%80s", str);
```

If the input is too long, it will be truncated.
We cannot know in advance how big the array might need to be.

To solve this problem we can use `realloc` to continuously resize an array while reading in one `char` at a time.

Strings of unknown length

Consider a function that repeatedly re-allocates memory, and returns a pointer to an arbitrarily long string:

Exercise

Poll 1: If n is the number of bytes read, what is the running time of `readstr`?

- (A) $O(1)$
- (B) $O(n)$
- (C) $O(n \log n)$
- (D) $O(n^2)$
- (E) $O(2^n)$

```c
// readstr() reads non-whitespace string from stdin
//    or returns NULL if unsuccessful
// effects: allocates memory [caller must free]
char *readstr(void) {
  char c;
  if (scanf(" %c", &c) != 1) return NULL;
  int len = 1;
  char *str = malloc(len * sizeof(char));
  str[0] = c;
  while (1) {
    if (scanf("%c", &c) != 1
        || 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;
}
```

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.

Instead, when we run out of space, we will allocate **twice as much** as we need. Then most of the time, we don't need to allocate any more.

We need to keep track of the "actual" length in addition to the *allocated* length.

Amortized Cost

```
char *readstr(void) {
  char c;
  if (scanf(" %c", &c) != 1) return NULL;
  int maxlen = 1;
  int len = 1;
  char *str = malloc(maxlen * sizeof(char));
  str[0] = c;
  while (1) {
    if (scanf("%c", &c) != 1
        || c == ' ' || c == '\n') break;
    if (len == maxlen) { // double!
      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;
}
```

Suppose we enter `cryptocurrency`.

Read	`len`	`maxlen`	
'c'	1	1	
'r'	2	1	double
'y'	3	2	double
'p'	4	4	
't'	5	4	double
'o'	6	8	
'c'	7	8	
'u'	8	8	
'r'	9	8	double
'r'	10	16	
'e'	11	16	
'n'	12	16	
'c'	13	16	
'y'	14	16	

Amortized Cost

With our "doubling" strategy, most iterations are $O(1)$.
Each time we `realloc` it costs $O(n)$, but this happens rarely.

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

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

> To read n values costs $O(n)$. The "average" or **amortized** cost is $O(n)/n = O(1)$.

You will use further *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 `push`es and `pop`s.

Languages that have a built-in resizable array (e.g. C++'s vector) often use a similar "doubling" strategy.

Opaque structures for ADTs

With dynamic memory, we can better implement Abstract Data Types.

Previously, we declared each `struct` in the interface file, creating a **transparent** structure:

```
struct rational {
  int numerator;
  int denominator;
};
```

Then the user could interact directly with the fields. They did not have to use the ADT, so they could corrupt it.

Opaque structures for ADTs

To make an **opaque** structure:

In the interface file we make a **forward declaration** of the `struct`:

```
// rational-secret.h
struct rational;
```

We also need to declare functions to create and clean up a `struct rational`:

```
// make_rat(n, d) Return a pointer to a new
// struct rational that stores n / d.
struct rational *make_rat(int n, int d);

// destroy_rat(rat) Clean up rat.
void destroy_rat(struct rational *rat);
```

The user knows that a `struct rational` exists, but not how big it is or anything about its fields.

In the implementation file, we declare the structure, and define the functions:

```
// rational-secret.c
struct rational { int n; int d; };

struct rational *make_rat(int n, int d) {
  assert(d != 0);
  struct rational *rat =
    malloc(sizeof(struct rational));
  rat->n = n;
  rat->d = d;
  return rat;
}
void destroy_rat(struct rational *rat) {
  assert(rat);
  free(rat);
}
```

Implementing a Stack ADT

We can now modify our `stack` module to be opaque, ensuring that users interact with it only through the ADT interface.

Exercise

- Move the declaration of `struct stack` from `stack.h` to `stack.c`.
- Modify `stack.h` so it has only the following forward declaration of `struct stack`:

```
struct stack;
```

- Add functions:
  - `struct stack *stack_create(void);` to **create** an empty stack,
  - `void stack_destroy(struct stack *s);` to **destroy** a stack.

Exercise

One you have tested your `stack` module, reimplement it so it starts a data size of 1, and uses the doubling strategy to be able to store an arbitrarily large number of items.

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