Static type system:

All types must be known before the program is run and the type of an indentifier cannot change.

C uses a static type system.

Uninitialized **global** variables are set to zero; **local** variables have undefined behaviour.

Function terminology

We **call** a function by **passing** it **arguments**. A function **returns** a value.

Entry point:

Where an operating system starts running a program. In C, the entry point is the special function `main`

Every C program has one and only one `main` function.

`main` has no parameters and an int return type: a successful program returns `0`.

Booleans:

Bools produce either `0` for `false` or `1` for `true`.

Any non-zero value is interpreted as `true`. Only zero or NULL are `false`.

C short-circuits boolean expressions.

Statements:

1. **Compound statements** (blocks {})
   a sequence of statements to be executed in order
2. **Expression statements**
   for generating side effects (return values are discarded)
3. **Control flow statements**
   control the order in which statements are executed
   (a) `return`
   (b) `if`, `else if`, `else`
   (c) `for`, `while`
   (d) function calls

Functional programming:

1. Functions only return values.
2. Return values only depend on arguments.
3. Only constants are used.

Function contracts:

1. **purpose statement**
2. **notes:** (optional)
3. **requires:** (communicate any non-asserted requirements with `[not asserted]`)
4. **effects:**
   (a) produces output
   (b) reads input
   (c) modifies a global variable/variable through a pointer parameter
   (d) allocates memory (caller must free)
   (e) frees ptr (ptr is now invalid)
5. **time:** specify what any variable refers to

Memory:

The smallest accessible unit of memory is a byte (8 bits). To access, its **address** is needed.

With n bytes of RAM, there are n memory addresses of storage from 0 up to $n-1$.

With n-bit addresses, there are 2^n addresses.

A variable is a name for a memory address. When an identifier appears in an expression, C fetches the contents stored at the address of the identifier.

Type sizes:

1. `int` : 4
2. `float` : 4
3. `double` 8
4. `char` : 1
5. `ptr` : 8
6. `array` : sizeof type * array length
7. stack frame return addresses: 8

Structures:

Definition:

```
struct posn {  // name of struct
    int x;     // types and names of fields
    int y;
};             // mandatory semicolon
```

Initialization:

```
    struct posn p = {3, 4};
    struct posn p = {.x = 3, .y = 4}; // equivalent
```

Any omitted fields are automatically set to 0.

Structs can be copied from struct to struct, but not re-assigned directly:

`s = {2, 3}; // INVALID`, `s = s2; // VALID`

Structs cannot be compared with `==`

Operators:

1. `varname.fieldname` (selection)
2. `varname->fieldname` (indirection selection)

   equivalent to `(*varname).field`

   used when `varname` is a pointer to a struct to select field values.

1. No memory is reserved for a structure definition: memory is only reserved when a `struct` variable is defined.
2. The size of a struct is at least the sum of the size of each field, but may be larger.
3. Always pass structures as arguments through pointers
   (a) Avoids copying the entire struct
   (b) Allows mutation of the struct

Opaque structures:

A black box that the client cannot see inside of, implemented with incomplete declarations: `struct box;`

If we want a transparent structure, simply put the complete definition of the struct in the interface file (.h file)

With an incomplete declaration, only pointers to an already created structure can be defined.

The module must provide a way to create and destroy structures.

Sections of Memory:

From low to high areas in memory:

- Code
  - Machine code, function addresses
- Read-Only Data
  - Global constants, string literals
- Global Data
  - Global mutable variables
- Heap
  - A pool of memory available to a program that can be allocated upon request. Allocated memory must be freed, and can be reused for a future allocation. If too much memory has already been allocated, attempts to borrow more will fail.
  - Grows towards the stack.
- Stack
  - Stores the call stack: a LIFO system of stack frames created by function calls.
  - Grows towards the heap.

The read-only and global data sections are created and initialized at compile time. All initial values are known, but variables are only in scope after their definition.

Stack Frames:

Each function call creates a stack frame that includes:

1. The argument values
2. All local variables (both mutable and constant, irrespective of scope: definitions that have not yet been reached have a value of `???`)
3. The return address (program location to return to e.g. `main:56`)

The return address for `main` is `OS`.

All stack frames have one and only one return address.

Pass by value:

Before functions can execute, all arguments must be values. C makes a copy of each argument value and places the **copy** into the stack frame.

Aliasing:

When the same memory address can be accessed from more than one variable.

I/O functions:

Format specifiers:

1. %d: integer
2. %f: float
3. %lf: double (long float)
4. %c: char
5. %s: string
6. %p: pointer
7. %%: the % symbol

printf:

returns the total number of printed characters

scanf:

returns the number of fields that were successfully converted and assigned or `EOF` if the end of file has been reached.

scanf requires a pointer to the field: `scanf("%d", &i) // read int, store at i`

Use `scanf("%c")` to read whitespace and `scanf(" %c")` to ignore whitespace.

const with pointers

1. int *p
   // p points to a mutable int
2. const int *p
   // p points to a constant int (cannot mutate through p, but possibly otherwise mutable)
3. int * const p
   // p constantly points to a mutable int
4. const int * const p
   // p constantly points to a constant int

Modules:

A module is a file that provides a collection of functions that share a common aspect or purpose.

A client requires the functions that a module provides.

Programs can be made up of many modules, but there must be one "root" or **main file** that acts only as a client. This is the file that runs the program and defines `main`.

Motivation (RAM):

1. **Re-usability:** A good module can service many different clients for many different purposes.
2. **Abstraction:** To use a module, a client only needs to know *what* it does, not how it does it.
3. **Maintainability:** it is far easier to test and debug a single module than a large program. If a bug is found, only the module containing the bug needs to be fixed.

Interface:

Interface files (.h extension) are also known as **h**eader files and include:

1. An overall description of the module
2. A function declaration for each provided function
3. Documentation (a purpose) for each provided function

Implementation:

The code of a module (e.g. function definitions). It is good style to `#include` the interface file within the implementation.

The interface is everything a client needs to use the module. The interface is provided to the client, and the implementation is hidden from the client.

Declarations vs definitions:

A declaration introduces an identifier into a program and specifies its type.

Function declaration: `int my_func(int a, int b);`

Variable declaration: `extern int g;`

A definition instructs C to "create" the identifier

Declarations extend the scope of an identifier, and can bring an identifier from another file into scope.

Scope:

Local/block scope:

Only available inside the block it is defined in.

Global scope:

1. **Program scope:** By default, C global identifiers are accessible to every file in the program if the file declares the identifier.
2. **Module scope:** Prefixing a definition with `static` hides a global identifier from other files in the program. It restricts the scope of a global identifier to the file it is defined in. They can no longer be accessed by other files that try to declare the identifier.

Standard modules:

1. stdio.h provides `printf`, `scanf`
2. assert.h provides `assert`
3. limits.h provides `INT_MIN`, `INT_MAX`
4. stdbool.h provides the `bool` data type and `true` and `false`
5. stdlib.h provides `NULL`, `exit` (terminates the program), `EXIT_SUCCESS` (defined as 0), and `EXIT_FAILURE` (non-zero), `malloc`, `free`, and `qsort`.
6. string.h provides `strlen`, `strcmp`, `strcpy`, `strcat`

Module design:

High cohesion:

The functions within a module are related and working toward a common goal.

Low coupling:

Little interaction between modules.

Information hiding:

The interface is designed to hide any implementation details from the client

1. **Security:** We may want to prevent the client from tampering with data used by the module. Even if the tampering is not malicious, we may want to ensure that the only way the client can interact with the module is through the interface. We may need to protect the client from themselves.
2. **Flexibility:** By hiding the implementation details from the client, we gain the flexibility to change the implementation in the future.

Abstract Data Types (ADTs):

Modules that only allow access to stored data through interface functions (ADT operations).

Collection ADTs:

An ADT designed to store an arbitrary number of items.

1. **Dictionary:**
   (a) lookup: returns the value associated with the given key
   (b) insert: adds a new key/value pair (or replaces an existing one)
   (c) remove: deletes a key/value pair
2. **Stack:**
   (a) push: adds an item to the top of the stack
   (b) pop: removes the top item of the stack
   (c) top: returns the top item of the stack
   (d) is_empty: determines if the stack is empty
3. **Queue:**
   (a) add_back: adds an item to the end of the queue
   (b) remove_front: removes the item at the front of the queue
   (c) front: returns the item at the front of the queue
   (d) is_empty: determines if the queue is empty
4. **Sequence:**
   (a) item_at: returns the item in the given position
   (b) insert_at: inserts a new item at the given position
   (c) remove_at: removes an item at the given position
   (d) length: returns the number of items in the sequence

We can implement ADTs with various data structures (e.g. association list or BST for a dictionary).

Regardless, the client would never know the implementation details.

Arrays:

Initialization: `int a[6] = {1, 2, 3, 4}` ; uninitialized entries are set to 0 provided at least one entry is initialized for local arrays (instead of undefined behaviour).

C supports automatic length declaration:

```
int a[] = {1, 2, 3, 4, 5, 6} // length 6
```

If used with string shorthand, C automatically null-terminates:

```
char a[] = "cat" // length 4 array equivalent to {'c', 'a', 't', '\0'}
```

Once defined, the array cannot be mutated at once; only individual elements can be mutated.

Aside from sizeof and `&a` versus `&p` , an array is essentially a constant pointer.

When an array is passed to a function, it decays to a pointer:

`int sum_array(int a[], int len)` is equivalent to `int sum_array(int *a, int len)`

We can represent 2D arrays by mapping them down to 1D:

```
int a[6] = {1, 2, 3, 4, 5, 6}; // 2 x 3 matrix
```

To access row i, column j, we access `a[i * NUMCOLS + j]` .

Instead of a 2D array of chars, it is more common to use an array of pointers, each pointing to a string literal:

```
char *aos[] = {"array", "of", "literals"};
```

Mutable strings are more awkward; define each one separately.

Oversized arrays circumvent the requirement to know the length in advance, but

1. They are wasteful if the maximum length is excessively large.
2. They are restrictive if the maximum length is too small.
3. We need to keep track of
   (a) The "actual" length of the array
   (b) The maximum length of the array
4. If we exceed the maximum, we can do several things:
   (a) The array is not modified and an error is displayed
   (b) A special return value is used
   (c) An assertion fails (program termination)
   (d) The program terminates with an explicit error message

Pointer Arithmetic:

With integers, arithmetic is performed as `p + i` $\implies$ `p + i * sizeof(*p)` to determine the new address that `p` stores.

Two pointers of the same type can be subtracted: if `q = p + i`, then `i = q - p`.

Two pointers cannot be added.

Array indexing syntax is a pointer arithmetic operator:

`a[i]` $\iff$ `*(a + i)` $\iff$ `*(i + a)` $\iff$ `i[a]`

In array pointer notation, square brackets are not used:

```
    for (const int *p = a; p < a + len; ++p) { // pointer arithmetic
        sum += *p;
    }

    for (int i = 0; i < len; ++i) {            // array indexing
        sum += a[i];
    }
```

Big O notation:

$O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq (n^3) \subseteq \cdots \subseteq (2^n)$

Selection sort:

The smallest element is selected to be the first element in the new sorted sequence, and then the next smallest element is selected to be the second element, and so on.

```
void selection_sort(int a[], int len) {
    int pos = 0;
    for (int i = 0; i < len - 1; ++i) {
        pos = i;
        for (int j = i + 1; j < len; ++j) {
            if (a[j] < a[pos]) {
            pos = j;
            }
        }
        swap(&a[i], &a[pos]);
    }
}
```

Insertion sort:

We consider the first element to be a sorted sequence (of length one).

We then "insert" the second element into the existing sequence into the correct position, and then the third element, and so on.

For each iteration of Insertion sort, the first i elements are sorted.

We then "insert" the element a[i] into the correct position, moving all of the elements greater than a[i] one to the right to "make room" for a[i].

```
void insertion_sort(int a[], int len) {
    for (int i = 1; i < len; ++i) {
        for (int j = i; j > 0 && a[j - 1] > a[j]; --j) {
            swap(&a[j], &a[j - 1]);
        }
    }
}
```

Quicksort:

Select the first element of the array as our "pivot"

Move all elements that are larger than the pivot to the back of the array

Move ("swap") the pivot into the correct position

Recursively sort the "smaller than" sub-array and the "larger than" sub-array

> The core quick sort function has parameters for the range of elements (first and last) to be sorted, so a wrapper function is required.

```
void quick_sort_range(int a[], int first, int last) {
    if (last <= first) return; // length is <= 1
    int pivot = a[first];      // first element is the pivot
    int pos = last;            // where to put next larger
    for (int i = last; i > first; --i) {
        if (a[i] > pivot) {
            swap(&a[pos], &a[i]);
            --pos;
        }
    }
    swap(&a[first], &a[pos]); // put pivot in correct place
    quick_sort_range(a, first, pos - 1); // sort smaller
    quick_sort_range(a, pos + 1, last);  // sort larger
}
```

Merge sort:

Split the array in half and sort each half.

```
// 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 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);
}
```

Sorting Algorithm Efficiencies:

	Best	Worst
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$
Quicksort	$O(n \log n)$	$O(n^2)$
Mergesort	$O(n \log n)$	$O(n \log n)$

Strings:

There is no built-in string type in C. By convention, strings are `char` arrays terminated by a null character: `'\0'` or `0`.

The null character is not `'0'`.

printf prints a string until the null terminator, and does not print the null terminator. scanf reads until a whitespace character

If there is not enough space reserved for scanf to read to (or in general), **buffer overflow** occurs: C goes past `'\0'` and continues writing into memory it shouldn't overwrite.

C strings in quotations (e.g., "string") that are in an expression (i.e., not part of an array initialization) are known as **string literals**. For each string literal, a null-terminated const char array is created in the read-only data section. In the code, the occurrence of the string literal is replaced with the address of the corresponding array.

string.h functions:

strlen

Returns the length of the string (i.e. the number of characters preceding but not including the first null terminator).

strcmp

`strcmp(s1, s2)` returns 0 if s1 and s2 are identical, negative if s1 is before s2 and positive if s1 is after s2 (since the return is `s1[i] - s2[i]`).

Never use `==` to compare strings: this compares addresses

strcpy

`strcpy(dest, src)` overwrites the contents of `dest` with the contents of `src`.

strcat

`strcat(dest, src)` appends/concatenates `src` to the end of `dest`.

Always ensure the `dest` array is large enough (including the null terminator) to avoid buffer overflow.

Advantages of Heap-Allocated Memory:

1. **Dynamic:** The allocation size can be determined at runtime
2. **Resizable:** A heap allocaton can be resized.
3. **Duration:** Heap allocations persist until they are freed. A function can create a data structure that continues to exist after it returns.
4. **Safety:** If memory runs out, it can be detected and handled properly (unlike stack overflows).

malloc:

`malloc(n)` requests n contiguous bytes of memory from the heap and returns a pointer to a block of n bytes, or `NULL` if not enough memory is available.

time: $O(1)$.

The heap memory provided by malloc is uninitialized.

Always use `sizeof` to improve communication:

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

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

It is good style to check every malloc return value to handle `NULL` values. However, unless otherwise specified, this check is not needed.

free:

free(p) returns the memory at p back to the heap

requires: p is from a previous malloc

effects: the memory at p is invalid

time: O(1)

Once p is freed, p is a **dangling pointer**: it is left unchanged and now points to invalid memory.

It is good style to assign a pointer `NULL` after freeing it.

It is invalid to free memory that was not allocated by malloc or that has already been freed. `free(NULL)` is fine and does nothing.

Memory Leak:

Allocated memory is not freed. Programs may suffer degraded performance or eventually crash.

Garbage Collection:

A garbage collector detects when memory is no longer in use and automatically frees it and returns it to the heap. Garbage collectors may be slow and affect program performance. C does not have a garbage collector, but Racket does.

realloc:

We can resize arrays by:

1. creating a new array
2. copying the contents of the old array to the new array
3. freeing the old array
4. assigning a pointer to the new array

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 is from a previous malloc/realloc, or p is `NULL` (then realloc is identical to malloc)

effects: the memory at p is invalid

time: $O(n)$ where $n = \min\{\text{oldsize}, \text{newsize}\}$

The address passed to realloc is invalid after the function returns.

Amortized Analysis:

Can be useful when an algorithm has expensive operations that are executed only rarely.

Distributes the runtime of a long, rare operation across the runtime of short, frequent operations.

`stack_push` only needs to resize on powers of two, so is $O(n)$ on only these calls and is $O(1)$ for all others. We can distribute these $O(n)$ calls across each call until the next $O(n)$ call, so each call is $O(2) = O(1)$ on average.

Linked Lists:

Each **node** contains an **item** and a **link** (pointer) to the next node in the list.

```
1  struct llnode {
2      int item;
3      struct llnode *next;
4      // struct llnode *prev;
5  };
6
7  struct llist {
8      struct llnode *front;
9      // struct llnode *back;
10     // int len;
11 };
```

Caching:

If we want to keep track of potentially helpful information, we can augment the `struct llist` to contain more fields.

We could also maintain a pointer to the back of the list to add to the back in $O(1)$ time:

```
struct llist {
    struct llnode *front;
    struct llnode *back;
    int length;
};
```

Caching introduces new ways that the structure and its data may be corrupted. The `length` field might not accurately reflect the true length.

Whenever the same information is stored in multiple ways, it is susceptible to *integrity* (consistency) issues.

If data integrity is an issue, we can exercise security by only providing interface functions to a client to protect them from themselves.

Node augmentations:

Each node is augmented to include more information about the node or the structure.

1. Dictionary: key and value
2. Priority Queue: item and priority of item
3. Doubly Linked List: next node and previous node
4. Trees: item and count (number of nodes in subtree containing the node: leaves have 1)

Trees:

In a binary tree, each node has at most two children. The height of an empty tree is zero.

```
struct bstnode {
    int item;
    struct bstnode *left;
    struct bstnode *right;
    // int count;
};

struct bst {
    struct bstnode *root;
}
```

The runtime of many binary search tree functions is $O(h)$ rather than $O(n)$. In an **unbalanced** tree, $h \approx n$, and in a **balanced** tree, $h \approx \log(n)$. A self-balancing tree re-arranges nodes during inserts and removals to ensure the tree is always balanced.

Queue functions:

Assume a doubly-linked list in all the following:

```
void queue_add_back(const int i, struct queue *q) {
    struct llnode *newnode = malloc(sizeof(struct queue);
    // handle failed malloc
    newnode->item = i;
    newnode->next = NULL;
    newnode->prev = q->back;
    if (q->front = NULL) {
        q->front = newnode;
    } else {
        q->back->next = newnode;
    }
    q->back = newnode;
    // ++q->length;
}
```

Remember the empty queue case and adding to the list

```
int queue_remove_front(struct queue *q) {
    assert(q->front);
    const int retval = q->front->item;
    struct queue *destroy = q->front;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->back = NULL;
    } else {
        q->front->prev = NULL;
    }
    free(destroy);
    // --q->len;
    return retval;
}
```

Remember the length 1 case and to set `q->front->prev = NULL`

BST functions:

A wrapper is required for a `struct bst`, the core function is:

```
void insert_bstnode(const int i, struct bstnode* node) {
    if (i < node->item) {
        if (node->left) {
            insert_bstnode(i, node->left);
        } else {
            node->left = new_leaf(i);
        }
    } else if (i > node->item) {
        if (node->right) {
            insert_bstnode(i, node->right);
        } else {
            node->right = new_leaf(i);
        }
    } // else do nothing, item already in
}
```

Dictionary functions:

```
void dict_insert(const int key, const char *val, struct dict *d) {
    struct bstnode *node = d->root;
    struct bstnode *parent = NULL;
    // locate insertion point
    while (node && node->item != key) {
        parent = node;
        if (key > node->item) {node = node->left;}
        else {node = node->right;}
    }
    if (node != NULL) { // key already exists
        free(node->value);
        node->value = my_strdup(val);
    } else if (parent == NULL) { // empty tree
        d->root = new_leaf(key, val);
    } else if (key < parent->item) {parent->left = new_leaf(key, val);}
    else {parent->right = new_leaf(key, val);}
}

void dict_remove(const int key, struct dict *d) {
    struct bstnode *target = d->root;
    struct bstnode *parent = NULL;
    // locate removal point
    while (target && key != target->item) {
        parent = target;
        if (key < target->item) {
            target = target->left;
        } else {
            target = target->right;
        }
    }
    if (target == NULL) return; // key not found
    // key was found, find replacement node
    struct bstnode *rep = NULL;
    if (target->left = NULL) {
        rep = target->right;
    } else if (target->right = NULL) {
        rep = target->left
    } else { // we have two children, take smallest node in right subtree
        // find replacement and its parent
        rep = target->right;
        struct bstnode *rep_parent = target;
        while (rep->left) {
            rep_parent = rep;
            rep = rep->left;
        }
        // update rep to be where target is
        rep->left = target->left;
        if (rep_parent != target) {
            rep->parent->left = rep->right;
            rep->right = target->right;
        }
        // free target and update its parent to point to rep
        free(target->value);
        free(target);
        if (parent == NULL) { // empty tree
            d->root = rep;
        } else if (key > parent->item) {
            parent->right = rep;
        } else {
            parent->left = rep;
        }
    }
}
```

Efficiencies:

These are all very intuitive:

Function	Dynamic Array	Linked List	Sorted DA	Sorted LL	Unb BST	Bal BST
item_at	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$#
search	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(\log n)$
insert_at	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
insert_front	$O(n)$	$O(1)$				
insert_back	$O(1)$*	$O(1)$†				
remove_at	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
remove_front	$O(n)$	$O(1)$				
remove_back	$O(1)$	$O(1)$‡				

*amortized

†requires a back pointer ($O(n)$ without)

‡requires a back pointer and doubly linked list ($O(n)$ without)

#requires a count augmentation ($O(n)$ without)

Void pointers:

void pointers can point at any type of data.

void pointers cannot point at functions.

void pointers cannot be dereferenced.

However, the address stored in a void pointer can be assigned to a pointer of the right type and then dereferenced.

We lose type safety, and could have a pointer pointing to the wrong type. Dereferencing is then UB.

Generic functions:

Void pointers are useful for handling any type of data:

```
1 void qsort(void *arr, const int len, size_t size,
2            int (*compare)(const void *, const void *));
```

`qsort` arguments are: the array, its length, the size of each element in the array and a generic comparison function.

`compare(a, b)` returns:

- negative if *a is before *b
- 0 if *a equals *b
- positive if *a is after *b

To write a generic map, we can pretend the array is a `char` array and perform pointer arithmetic manually using the `size` parameter (since `char`s are 1 byte)

Memory Management with Generic ADTs:

A generic ADT does not know the types of items it is dealing with, so it does not know the type of memory those items use, for instance:

1. non-heap memory (e.g. string literal)
2. a heap allocation (e.g. dynamic array)
3. a structure of many heap allocations (e.g. linked list)

Consequently, the ADT cannot perform any memory management on its items (naive `free` calls). If we simply free the overarching data structure, this would cause memory leaks if its elements are also heap-allocated.

We can deal with this in two ways:

1. Require the collection to be empty before `free`ing the overarching data structure.
2. Require the client to provide an appropriate `free` function.

   This requires caching a pointer to a free function in the ADT structure that the client provides upon creation.

```
struct stack {
    int len;
    int maxlen;
    void **data;
    void (*free_item)(void *); // function pointer
}

```

   This is then used in the `destroy` function.

It is important to design a clear interface to communicate to the client how the memory for the individual items will be managed.

Sequenced data:

The original sequencing of data (order it was inputted in) is important (e.g. names in order of competition place). A data structure that sorts the data is likely not appropriate.

- An array is good if you frequently access elements at specific positions (random access).
- A linked list is good if you frequently add and remove elements at the start.

For unsequenced data, the original sequencing is unimportant (e.g. list of party guests).

- A self-balancing BST is good if you frequently search for, add and remove items.
- A sorted array is good if you rarely add/remove elements, but frequently search for elements and select the data in sorted order