Arrays

Optional Textbook Readings: CP:AMA 8.1, 9.3, 12.1, 12.2, 12.3

The primary goal of this section is to be able to use arrays.

Arrays

C only has two *built-in* types of “compound” data storage:

- `struct`ures
- ***arrays***

```
int my_array[6] = {4, 8, 15, 16, 23, 42};
```

An array is a data structure that contains a **fixed number** of elements that all have the **same type**.

Because arrays are *built-in* to C, they are used for many tasks where *lists* are used in Racket, but **arrays and lists are very different**. In Section 11 we construct Racket-like lists in C.

```
int my_array[6] = {4, 8, 15, 16, 23, 42};
```

To define an array we must know the **length** of the array **in advance** (we address this limitation in Section 10).

Each individual value in the array is known as an ***element***. To access an element, its ***index*** is required.

The first element of `my_array` is at index `0`, and it is written as `my_array[0]`.

The second element is `my_array[1]` and the last is `my_array[5]`.

In computer science we often start counting at `0`.

example: accessing array elements

Each individual array element can be used in an expression as if it was a variable.

In addition, the index of the array can be an expression.

```
int a[6] = {4, 8, 15, 16, 23, 42};

int j = a[0];            // j is 4
int *p = &a[j - 1];      // p points at a[3]

a[2] = a[a[0]];          // a[2] is now 23
++a[1];                  // a[1] is now 9
```

example: arrays & iteration

Arrays and iteration are a powerful combination.

```
int a[6] = {4, 8, 15, 16, 23, 42};
int sum = 0;

for (int i = 0; i < 6; ++i) {
  printf("a[%d] = %d\n", i, a[i]);
  sum += a[i];
}
printf("sum = %d\n", sum);
```

```
a[0] = 4
a[1] = 8
a[2] = 15
a[3] = 16
a[4] = 23
a[5] = 42
sum = 108
```

Array initialization

Arrays can only be **initialized** with braces (`{}`).

```
int a[6] = {4, 8, 15, 16, 23, 42};

a = {0, 0, 0, 0, 0, 0};          // INVALID
a = ??? ;                        // INVALID
```

Once defined, the entire array cannot be mutated at once, and the length cannot change. Only *individual elements* can be mutated.

If there are not enough elements in the initialization braces, the remaining values are initialized to zero.

```
int b[5] = {1, 2, 3};       // b[3] & b[4] = 0
int c[5] = {0};             // c[0]...c[4] = 0
```

Character arrays can be initialized with double quotes (") for convenience.

The following two definitions are equivalent:

```
char a[3] = {'c', 'a', 't'};
char b[3] = "cat";
```

In this example, a and b are character arrays and are not valid strings. This will be revisited in Section 09.

Like variables, the value of an uninitialized array depends on the scope of the array:

```
int a[5];  // uninitialized
```

- uninitialized *global* arrays are zero-filled.
- uninitialized *local* arrays are filled with arbitrary (“garbage”) values from the stack.

Array length

C does not explicitly keep track of the array **length** as part of the array data structure.

You must keep track of the array length separately.

To improve readability, the array length is often stored in a separate variable.

```
int a[6] = {4, 8, 15, 16, 23, 42};
const int a_len = 6;
```

It might seem better to use a constant to specify the length of an array.

```
const int a_len = 6;
int a[a_len] = {4, 8, 15, 16, 23, 42};  // NOT IN CS136
```

This would appear to be a “better style”.

However, the syntax to do this properly is outside of the scope of this course (see following slide).

In this course, always define arrays using numbers. It is okay to have these “magic numbers” appear in your assignments.

```
int a[6] = ...;
```

Many programming guides recommend using the *unsigned* integer type `size_t` instead of an `int` to loop through an array.

```
for (size_t i = 0; i < a_len; ++i) { ... }
```

For example, array lengths may be greater than `INT_MAX`.

Because `size_t` is *unsigned*, you have to be careful when looping backwards through an array:

```
for (size_t i = a_len - 1; i >= 0; --i) { ... }
// infinite loop: i will never be negative
```

In this course we are not going to use advanced `int` types, including `size_t`.

A preferred syntax to specify the length of an array is to define a *macro*.

```
#define A_LEN 6

int main(void) {
  int a[A_LEN] = {4, 8, 15, 16, 23, 42};
  // ...
```

In this example, `A_LEN` is not a constant or even a variable.

`A_LEN` is a *preprocessor macro*. Every occurrence of `A_LEN` in the code is replaced with 6 before the program is run.

C99 supports *Variable Length Arrays* (VLAs), where the length of an **uninitialized** local array can be specified by a variable (or a function parameter) not known in advance. The size of the stack frame is increased accordingly.

```
int some_function(int n) {
  int m = n * 2;
  int a[m];      // length determined at run time
  // ...
```

This approach has many disadvantages and in more recent versions of C, this feature was removed (made optional). **You are not allowed to use VLAs in this course.** In Section 10 we see a better approach.

Theoretically, in some circumstances `sizeof` can be used to determine the length of an array.

```
  int len = sizeof(a) / sizeof(a[0]);
```

The CP:AMA textbook uses this on occasion.

However, in practice (and in this course) this should be avoided, as the `sizeof` operator only properly reports the array size in very specific circumstances.

Array size

The **length** of an array is the number of elements in the array.

The **size** of an array is the number of bytes it occupies in memory.

An array of k elements, each of size s, requires exactly $k \times s$ bytes.

In the C memory model, array elements are adjacent to each other. Each element of an array is placed in memory immediately after the previous element.

If `a` is an integer array with six elements (`int a[6]`) the size of `a` is: $(6 \times$ `sizeof(int)`$) = 6 \times 4 = 24$.

Not everyone uses the same terminology for length and size.

example: array in memory

```
int a[6] = {4, 8, 15, 16, 23, 42};
printf("&a[0] = %p ... &a[5] = %p\n", &a[0], &a[5]);
```

```
&a[0] = 0x5000 ... &a[5] = 0x5014
```

addresses	contents (4 bytes)
`0x5000 ... 0x5003`	4
`0x5004 ... 0x5007`	8
`0x5008 ... 0x500B`	15
`0x500C ... 0x500F`	16
`0x5010 ... 0x5013`	23
`0x5014 ... 0x5017`	42

The array identifier

An array does not have a “value” in C. When an array is used by itself in an expression, it evaluates (“decays”) to the **address** of the array (&a), which is also the address of the first element (&a[0]).

```
int a[6] = {4, 8, 15, 16, 23, 42};
trace_ptr(a);
trace_ptr(&a);
trace_ptr(&a[0]);
```

```
a => 0x5000
&a => 0x5000
&a[0] => 0x5000
```

Even though a and &a have the same *value*, they have different *types*, and cannot always be used interchangeably.

Dereferencing the array (*a) is equivalent to referencing the first element (a[0]).

```
int a[6] = {4, 8, 15, 16, 23, 42};

trace_int(a[0]);
trace_int(*a);
```

```
a[0] => 4
*a => 4
```

Passing arrays to functions

When an array is passed to a function, only the **address** of the array is copied into the stack frame.

This is more efficient than copying the entire array to the stack.

Typically, the length of the array is unknown to the function, and is a separate parameter.

There is no method of “enforcing” that the length passed to a function is valid.

Functions should **require** that the length is valid, but there is no way for a function to `assert` that requirement.

example: array parameters

```
int sum_array(int a[], int len) {
  int sum = 0;
  for (int i = 0; i < len; ++i) {
    sum += a[i];
  }
  return sum;
}

int main(void) {
  int my_array[6] = {4, 8, 15, 16, 23, 42};
  trace_int(sum_array(my_array, 6));
}
```

```
sum_array(my_array, 6) => 108
```

Note the parameter syntax: `int a[]`

and the calling syntax: `sum_array(my_array, 6)`.

example: “pretty” print an array

```c
// pretty prints an array with commas, ending with a period
// requires: len > 0

void print_array(int a[], int len) {
  assert(len > 0);
  for (int i = 0; i < len; ++i) {
    if (i) {
      printf(", ");
    }
    printf("%d", a[i]);
  }
  printf(".\n");
}

int main(void) {
  int a[6] = {4, 8, 15, 16, 23, 42};
  print_array(a, 6);
}
```

```
4, 8, 15, 16, 23, 42.
```

C allows you to specify the intended length of the array in the parameter, but it is **ignored**.

```
void calendar(int days_per_month[12]) {
  // ...
}
```

In this example, the `12` is ignored. The function may be passed an array of arbitrary length.

Similarly, some prefer to pass the length of the array first:

```
void f(int len, int a[len]) {
  // ...
}
```

But since the `[len]` is ignored (and not enforced) it is more common to pass the array first.

As we have seen before, passing an address to a function allows the function to change (mutate) the contents at that address.

```
void array_negate(int a[], int len) {
  for (int i = 0; i < len; ++i) {
    a[i] = -a[i];
  }
}
```

It's good style to use the `const` keyword to both prevent mutation and communicate that no mutation occurs.

```
int sum_array(const int a[], int len) {
  int sum = 0;
  for (int i = 0; i < len; ++i) {
    sum += a[i];
  }
  return sum;
}
```

Array within a structure

Because a structure can contain an array:

```
struct mystruct {
  int big[10000];
};
```

It is *especially* important to pass a pointer to such a structure, otherwise, the **entire array** is copied to the stack frame.

```
int very_slow(struct mystruct s) {
  ...
}

int much_faster(struct mystruct *s) {
  ...
}
```

Pointer arithmetic

We have not yet discussed any ***pointer arithmetic***.

C allows an integer to be added to a pointer, but the result may not be what you expect.

If `p` is a pointer, the value of `(p+1)` **depends on the type** of the pointer `p`.

`(p+1)` adds the `sizeof` whatever `p` points at.

According to the official C standard, pointer arithmetic is only valid **within an array** (or a structure) context. This becomes clearer later.

Pointer arithmetic rules

- When adding an integer `i` to a pointer `p`, the address computed by `(p + i)` in C is given in "normal" arithmetic by:

$$\mathtt{p} + \mathtt{i} \times \mathtt{sizeof}(*\mathtt{p}).$$

- Subtracting an integer from a pointer `(p - i)` works in the same way.
- Mutable pointers can be incremented (or decremented).
  `++p` is equivalent to `p = p + 1`.

- You cannot add two pointers.
- A pointer `q` can be subtracted from another pointer `p` if the pointers are the same type (point to the same type). The value of `(p-q)` in C is given in "normal" arithmetic by:

  $$(\mathtt{p} - \mathtt{q})/\mathtt{sizeof}(*\mathtt{p}).$$

  In other words, if `p = q + i` then `i = p - q`.
- Pointers (of the same type) can be compared with the comparison operators: <, <=, ==, !=, >=, > (*e.g.*, `if (p < q) ...`).

Pointer arithmetic and arrays

Pointer arithmetic is useful when working with **arrays**.

Recall that for an array `a`, the value of `a` is the address of the first element (`&a[0]`).

Using pointer arithmetic, the address of the second element `&a[1]` is (`a + 1`), and it can be referenced as `*(a + 1)`.

The array indexing syntax (`[]`) is an **operator** that performs *pointer arithmetic*.

`a[i]` is *equivalent* to `*(a + i)`.

C does not perform any array “bounds checking”.

For a given array `a` of length l, C does not verify that `a[j]` is valid ($0 \le j < l$).

C simply “translates” `a[j]` to `*(a + j)`, which may be outside the *bounds* of the array (*e.g.*, `a[1000000]` or `a[-1]`).

This is a common source of errors and bugs and a common criticism of C. Many modern languages have fixed this shortcoming and have “bounds checking” on arrays.

In ***array pointer notation***, square brackets ([ ]) are not used, and all array elements are accessed through pointer arithmetic.

```
int sum_array(const int *a, int len) {
  int sum = 0;
  for (const int *p = a; p < a + len; ++p) {
    sum += *p;
  }
  return sum;
}
```

Note that the above code behaves **identically** to the previously defined `sum_array`:

```
int sum_array(const int a[], int len) {
  int sum = 0;
  for (int i = 0; i < len; ++i) {
    sum += a[i];
  }
  return sum;
}
```

another example: pointer notation

```
// count_match(item, a, len) counts the number of
//   occurrences of item in the array a
int count_match(int item, const int *a, int len) {
  int count = 0;
  const int *p = a;
  while (p < a + len) {
    if (*p == item) {
      ++count;
    }
    ++p;
  }
  return count;
}
```

Remember, for the variable:

```
const int *p
```

you can mutate `p` but you cannot mutate `*p`.

The choice of notation (pointers or `[]`) is a matter of style and context. You are expected to be comfortable with both.

C makes no distinction between the following two function declarations:

```
int array_function(int a[], int len) {...}   // a[]
int array_function(int *a,  int len) {...}   // *a
```

In *most* contexts, there is no practical difference between an array identifier and an immutable pointer.

The subtle differences between an array and a pointer are discussed at the end of Section 09.

Array map

Aside from the awkward function pointer parameter syntax, the implementation of `array_map` is straightforward.

```
// array_map(f, a, len) replaces each element a[i]
//   with f(a[i])
// effects: modifies a

void array_map(int (*f)(int), int a[], int len) {
  for (int i = 0; i < len; ++i) {
    a[i] = f(a[i]);
  }
}
```

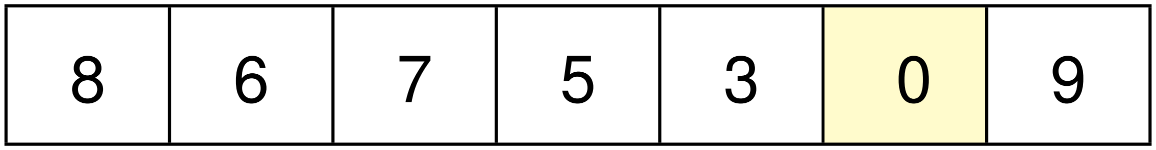
```c
#include "array_map.h"

int add1(int i) {
  return i + 1;
}

int sqr(int i) {
  return i * i;
}

int main(void) {
  int a[6] = {4, 8, 15, 16, 23, 42};
  print_array(a, 6);
  array_map(add1, a, 6);
  print_array(a, 6);
  array_map(sqr, a, 6);
  print_array(a, 6);
}
```

```
4, 8, 15, 16, 23, 42.
5, 9, 16, 17, 24, 43.
25, 81, 256, 289, 576, 1849.
```

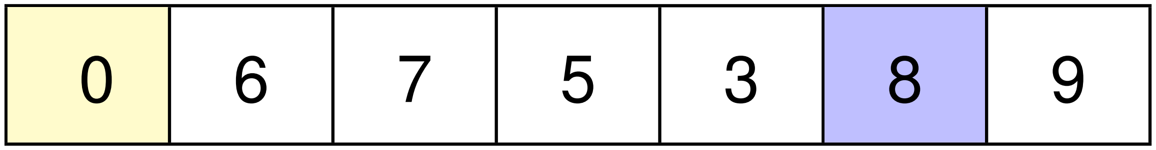
Selection sort

In ***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.

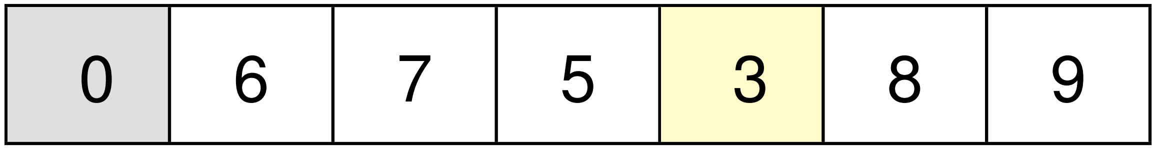
First, we find the position of the smallest element...



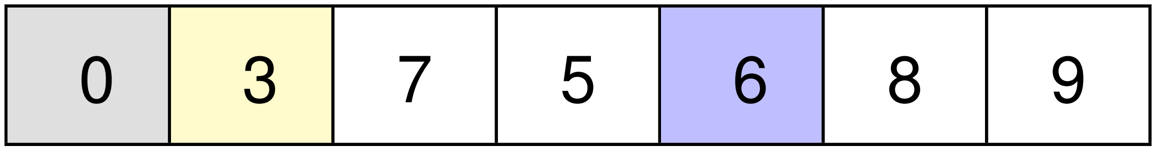
and then we *swap* the first element with the smallest.



Then, we find the next smallest element...



and then we *swap* that element with the second one, and so forth...



```
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]);  // see Section 05
  }
}
// Notes:
//   i:   loops from 0 ... len-2 and represents the
//        "next" element to be replaced
//   j:   loops from i+1 ... len-1 and is "searching"
//        for the next smallest element
//   pos: position of the "next smallest"
```

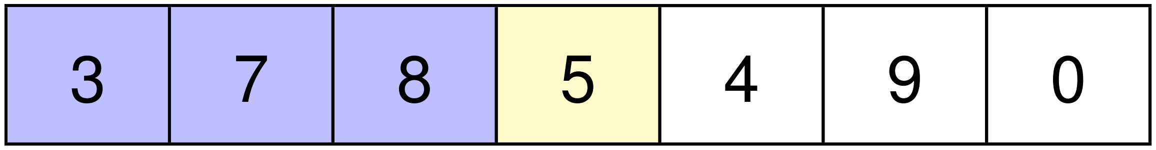
Insertion sort

In ***Insertion sort***, we consider the first element to be a sorted sequence (of length one).

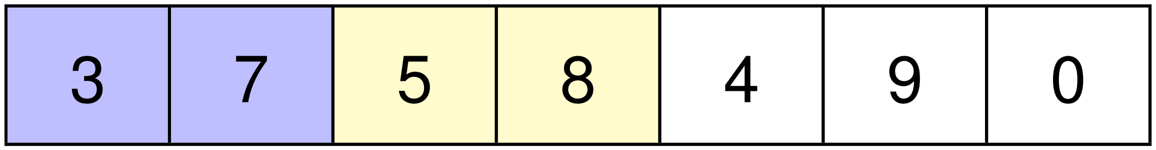
We then “insert” the second element into the existing sequence into the correct position, and then the third element, and so on.

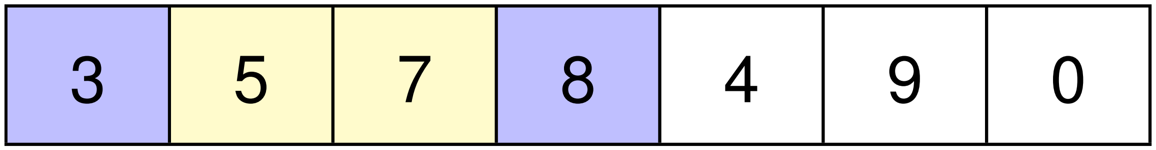
For each iteration of *Insertion sort*, the first `i` elements are sorted. We then “insert” the element `a[i]` into the correct position, moving all of the elements greater than `a[i]` one to the right to “make room” for `a[i]`.

Consider an iteration of insertion sort (`i = 3`), where the first `i` (3) elements have been sorted. We want to *insert* the element at `a[i]` into the correct position.

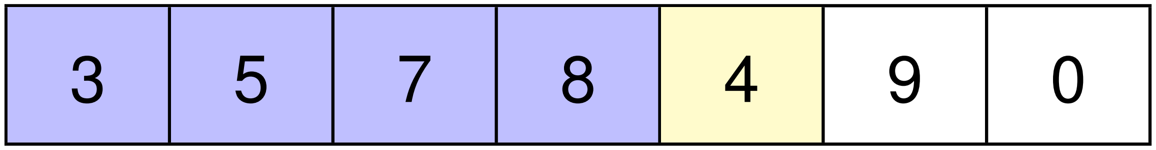


We continue to *swap* the element with the previous element until it reaches the correct position.





Once it is in the correct position, we start on the next element.



```
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]);
    }
  }
}

// Notes:
//   i:   loops from 1 ... len-1 and represents the
//        "next" element to be replaced
//   j:   loops from i ... 1 and is "inserting"
//        the element that was at a[i] until it
//        reaches the correct position
```

Quicksort

Quicksort is an example of a “divide & conquer“ algorithm.

First, an element is selected as a “pivot” element.

The list is then **partitioned** (*divided*) into two sub-groups: elements *less than* (or equal to) the pivot and those *greater than* the pivot.

Finally, each sub-group is then sorted (*conquered*).

Quicksort is also known as partition-exchange sort or Hoare’s quicksort (named after the author).

We have already seen the implementation of quick sort in racket.

```
(define (quick-sort lon)
  (cond [(empty? lon) empty]
    [else (define pivot (first lon))
          (define less (filter (lambda (x)
                              (<= x pivot)) (rest lon)))
          (define greater (filter (lambda (x)
                                 (> x pivot)) (rest lon)))
          (append (quick-sort less)
                  (list pivot)
                  (quick-sort greater))]))
```

For simplicity, we select the first element as the “pivot”. A more in-depth discussion of pivot selection occurs in CS 240.

In our C implementation of quick sort, we:

- 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 `quick_sort_range` 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);
  quick_sort_range(a, pos + 1, last);
}

void quick_sort(int a[], int len) {
  quick_sort_range(a, 0, len - 1);
}
```

Linear search

In Racket, the built-in function `member` can be used to determine if a list contains an element.

We can write a similar function in C that finds the index of an element in an array:

```
// find(item, a, len) finds the index of item in a,
//   or returns -1 if it does not exist

int find(int item, const int a[], int len) {
  for (int i = 0; i < len; ++i) {
    if (a[i] == item) {
      return i;
    }
  }
  return -1;
}
```

Binary search

If the array is sorted, we can use ***binary search***:

```
// requires: a is sorted in ascending order [not asserted]
int find_sorted(int item, const int a[], int len) {
  int low = 0;
  int high = len - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    if (a[mid] == item) {
      return mid;
    } else if (a[mid] < item) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return -1;
}
```

In Section 08 we will see this is more *efficient* than linear search.

Multi-dimensional data

All of the arrays seen so far have been one-dimensional (1D) arrays.

We can represent multi-dimensional data by “mapping” the higher dimensions down to one.

For example, consider a 2D array with 2 rows and 3 columns.

```
1 2 3
7 8 9
```

We can represent the data in a simple one-dimensional array.

```
int data[6] = {1, 2, 3, 7, 8, 9};
```

To access the entry in row `r` and column `c`, we simply access the element at `data[r*3 + c]`.

In general, it would be `data[row * NUMCOLS + col]`.

C supports multiple-dimension arrays, but they are not covered in this course.

```
int two_d_array[2][3];
int three_d_array[10][10][10];
```

When multi-dimensional arrays passed as parameters, the second (and higher) dimensions must be fixed.
(*e.g.*, `int function_2d(int a[][10], int numrows)`).

Internally, C represents a multi-dimensional array as a 1D array and performs “mapping” similar to the method described in the previous slide.

See CP:AMA sections 8.2 & 12.4 for more details.

Oversized Arrays

A significant limitation of an array is that the length of the array must be known **in advance**.

In Section 10 we introduce *dynamic memory* which can be used to circumvent this limitation, but first we explore a less sophisticated approach.

In some applications, it may be “appropriate” (or “easier”) to have an **oversized** array with a “maximum” length.

In general, oversized arrays should only be used when appropriate:

- They are wasteful if the maximum length is excessively large.
- They are restrictive if the maximum length is too small.

When working with oversized arrays, we need to keep track of

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

To illustrate oversized arrays, we implement an integer **stack** structure with a maximum length of 100 elements.

The `len` field keeps track of the *actual* length of the stack.

```
struct stack {
  int len;
  int maxlen;
  int data[100];
};
```

We need to provide a `stack_init` function to initialize the structure:

```
void stack_init(struct stack *s) {
  assert(s);
  s->len = 0;
  s->maxlen = 100;
}
```

Ignoring the `push` operation for now, we can write the rest of the stack 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 > 0);
  return s->data[s->len - 1];
}

// note: stack_pop returns the element popped
int stack_pop(struct stack *s) {
  assert(s);
  assert(s->len > 0);
  s->len -= 1;
  return s->data[s->len];
}
```

What happens if we exceed the maximum length when we try to `push` an element?

There are a few possibilities:

- the stack is not modified and an error message is displayed
- a special return value can be used
- an `assert`ion fails (terminating the program)
- the program explicitly **terminates** with an error message

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.

The argument passed to `exit` is equivalent to the `return` value of `main`.

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

```
if (something_bad) {
  printf("FATAL ERROR: Something bad happened!\n");
  exit(EXIT_FAILURE);
}
```

```
// stack_push(item, s) pushes item onto stack s
// requires: s is a valid stack
// effects:  modifies s
//           may display output and exit

void stack_push(int item, struct stack *s) {
  assert(s);
  if (s->len == s->maxlen) {
    printf("FATAL ERROR: stack capacity (%d) exceeded\n",
           s->maxlen);
    exit(EXIT_FAILURE);
  }
  s->data[s->len] = item;
  s->len += 1;
}
```

Goals of this Section

At the end of this section, you should be able to:

- define and initialize arrays
- use iteration to loop through arrays
- use pointer arithmetic
- explain how arrays are represented in the memory model, and how the array index operator ([ ]) uses pointer arithmetic to access array elements in constant time
- use both array index notation ([ ]) and array pointer notation and convert between the two

- use oversized arrays
- describe selection sort, insertion sort, quicksort and binary search on a sorted array
- represent multi-dimensional data in a single-dimensional array