Arrays & Strings

Readings: CP:AMA 8.1, 9.3, 12.1, 12.2, 12.3, 13

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.

```
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 assigned to at once.

Each *individual element* must be mutated.

```
for (int i=0; i < 6; ++i) {
  a[i] = 0;
}
```

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

```
int a[5];
```

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

If there are not enough elements in the braces, the remaining values are initialized to zero (even with local arrays).

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

If an array is initialized, the length of the array can be omitted from the declaration and *automatically* determined from the number of elements in the initialization.

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

This syntax is only allowed if the array is initialized.

```
int b[];  // INVALID
```

Similar to structures, C99 supports a partial initialization syntax.

```
int a[100] = { [50] = 1, [25] = -1, [75] = 3 };
```

Omitted elements are initialized to zero.

C99 allows the length of an **uninitialized local array** to be determined *while the program is running*. The size of the stack frame is increased accordingly.

```
int count;
printf("How many numbers? ");
scanf("%d", &count);

int a[count];       // count determined at run-time
```

This approach has many disadvantages and in the most recent version of C (C11), this feature was made optional. In Section 10 we see a better approach.

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: $(6 \times \texttt{sizeof(int)}) = 6 \times 4 = 24$.

> 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

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.

Typically, the array length is stored in a separate variable.

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

```
const int a_length = 6;
int a[a_length];
```

The above definition is fine in `Seashell`, but some C environments do not allow the length of the array to be specified by a variable.

In those environments, the `#define` syntax is more often used. This is common in CP:AMA.

```
#define A_LENGTH 6
int a[A_LENGTH];
```

Theoretically, in some circumstances you could use `sizeof` 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, this should be avoided, as the `sizeof` operator only properly reports the array size in very specific circumstances.

The array identifier

The **value** of an array (a) is the same as the **address** of the array (&a), which is also the address of the first element (&a[0]).

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

```
0x5000 0x5000 0x5000
4 4
```

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

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, and is provided as a separate parameter.

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};
  int sum = sum_array(my_array, 6);
}
```

Note the parameter syntax: `int a[]`

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

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

```
void array_add1(int a[], int len) {
  for (int i = 0; i < len; ++i) {
    ++a[i];
  }
}
```

It's good style to use the `const` keyword to 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;
}
```

Because a structure can contain an array:

```
struct mystruct {
  int big[1000];
};
```

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

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

int 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.

- You can subtract a pointer `q` 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:

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

  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)`.

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;
}
```

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 a *(constant)* pointer.

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

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);
  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.
```

Array map

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

```
// effects: replaces each element a[i] with f(a[i])

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[] = {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...

8	6	7	5	3	0	9

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

0	6	7	5	3	8	9

Then, we find the next smallest element...

0	6	7	5	3	8	9

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

0	3	7	5	6	8	9

```c
void selection_sort(int a[], int len) {
  for (int i=0; i < len - 1; ++i) {
    int 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"
```

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.

```c
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);
}
```

Binary 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 -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;
}
```

But what if the array was previously *sorted*?

We can use **binary search** to find the element faster:

```
int find_sorted(int item, const int a[], int len) {
  int low = 0;
  int high = len-1;
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (a[mid] == item) {
      return mid;
    } else if (a[mid] < item) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return -1;
}
```

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.

Fixed-Length Arrays

A significant limitation of an array is that you need to know the length of the array **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 a **maximum length** for an array.

In general, maximums should only be used when appropriate:

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

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

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

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

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

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

We will 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);
  return s->data[s->len - 1];
}

// note: stack_pop returns the element popped
int stack_pop(struct stack *s) {
  assert(s);
  assert(s->len);
  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` (`0`) and `EXIT_FAILURE` (non-zero).

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

```c
// stack_push(item, s) pushes item onto stack s
// requires: s is a valid stack
// effects:  modifies s
//           if max stack size is exceeded,
//           prints a message and exits

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

Strings

There is no built-in C ***string*** *type*. The "**convention**" is that a C string is an **array of characters**, terminated by a ***null character***.

```
char my_string[4]  = {'c', 'a', 't', '\0'};
```

The *null character*, also known as a null ***terminator***, is a `char` with a value of zero. It is often written as `'\0'` instead of just `0` to improve communication and indicate that a null character is intended.

`'\0'` (ASCII `0`) is different than `'0'` (ASCII 48), which is the character for the symbol zero.

String initialization

`char` arrays also support a double quote (`"`) **initialization** syntax.

When combined with the automatic length declaration (`[]`), the length includes the null terminator.

The following definitions create equivalent 4-character arrays:

```
char a[]  = {'c', 'a', 't', '\0'};
char b[]  = {'c', 'a', 't', 0};
char c[4] = {'c', 'a', 't'};
char d[]  = { 99,  97, 116, 0};
char e[4] = "cat";
char f[]  = "cat";
```

This array **initialization** notation is **different** than the double quote notation used in expressions (*e.g.*, in `printf("string")`).

Null termination

With null terminated strings, we do not have to pass the *length* to every function. It can be automatically determined.

```
// e_count(s) counts the # of e's and E's in string s

int e_count(const char s[]) {
  int count = 0;
  int i = 0;
  while (s[i]) {  // not the null terminator
    if ((s[i] == 'e')||(s[i] == 'E')) {
      ++count;
    }
    ++i;
  }
  return count;
}
```

As with “regular” arrays, it is good style to have `const` parameters to communicate that no changes (mutation) occurs to the string.

strlen

The `string` library (`#include <string.h>`) provides many useful functions for processing strings (more on this library later).

The `strlen` function returns the length of the *string*, **not** necessarily the length of the *array*. It does **not include** the null character.

```
int my_strlen(const char s[]) {
  int len = 0;
  while (s[len]) {
    ++len;
  }
  return len;
}
```

Here is an alternative implementation of `my_strlen` that uses pointer arithmetic.

```
int my_strlen(const char *s) {
  const char *p = s;
  while (*p) {
    ++p;
  }
  return (p-s);
}
```

Lexicographical order

Characters can be easily compared (`c1 < c2`) as they are numbers, so the character **order** is determined by the ASCII table.

If we try to compare two strings (`s1 < s2`), C compares their *pointers*, which is not helpful.

To compare strings we are typically interested in using a **lexicographical order**.

> Strings require us to be more careful with our terminology, as “smaller than” and “greater than” are ambiguous: are we considering just the **length** of the string? To avoid this problem we use **precedes** (“before”) and **follows** (“after”).

To compare two strings using a **lexicographical order**, we first compare the first character of each string. If they are different, the string with the smaller first character *precedes* the other string. Otherwise (the first characters are the same), the second characters are compared, and so on.

If the end of one string is encountered, it *precedes* the other string. Two strings are equal (the same) if the are the same length and all of their characters are identical.

The following strings are in lexicographical order:

```
"" "a" "az" "c" "cab" "cabin" "cat" "catastrophe"
```

The <`string.h`> library function `strcmp` uses lexicographical ordering.

`strcmp(s1, s2)` returns zero if the strings are identical. If `s1` precedes `s2`, it returns a negative integer. Otherwise (`s1` follows `s2`) it returns a positive integer.

```
int my_strcmp(const char s1[], const char s2[]) {
  int i = 0;
  while (s1[i] == s2[i]) {
    if ((s1[i] == '\0') && (s2[i] == '\0')) return 0;
    ++i;
  }
  if (s1[i] < s2[i]) return -1;
  return 1;
}
```

To compare if two strings are *equal* (identical), use the `strcmp` function.

The equality operator (==) only compares the *addresses* of the strings, and not the contents of the arrays.

```
char a[] = "the same?";
char b[] = "the same?";
char *s = a;

if (a == b) ...              // False (diff. addresses)
if (strcmp(a, b) == 0) ... // True  (proper comparison)
if (a == s) ...              // True  (same addresses)
```

Lexicographical orders can be used to compare (and sort) any *sequence* of elements (arrays, lists, ...) and not just strings.

The following Racket function lexicographically compares two lists of numbers:

```
(define (lon<=? lon1 lon2)
  (cond [(empty? lon1) #t]
        [(empty? lon2) #f]
        [(< (first lon1) (first lon2)) #t]
        [(< (first lon2) (first lon1)) #f]
        [else (lon<=? (rest lon1) (rest lon2))]))

(lon<=? '(4 9 1 2 1) '(4 5 9)) ; => #f
(lon<=? '(4 3) '(4 3 2))       ; => #t
```

String I/O

The `printf` placeholder for strings is `%s`.

```
char a[] = "cat";
printf("the %s in the hat\n", a);
```

`printf` prints out characters until the null character is encountered.

When using `%s` with `scanf`, it stops reading the string when a “white space” character is encountered (*e.g.,* a space or `\n`).

`scanf("%s")` is useful for reading in one “word” at a time.

```
char name[81];
printf("What is your first name?\n");
scanf("%s", name);
```

You must be very careful to reserve enough space for the string to be read in, and **do not forget the null character**.

In this example, the array is `81` characters and can accommodate first names with a length of up to `80` characters.

What if someone has a *really* long first name?

example: scanf

```
int main(void) {
  char command[8];
  int balance = 0;
  while (1) {
     printf("Command? ('balance', 'deposit', or 'q' to quit): ");
     scanf("%s", command);
     if (strcmp(command, "balance") == 0) {
       printf("Your balance is: %d\n", balance);
     } else if (strcmp(command, "deposit") == 0) {
       printf("Enter your deposit amount: ");
       int dep;
       scanf("%d", &dep);
       balance += dep;
     } else if (strcmp(command, "q") == 0) {
       printf("Bye!\n"); break;
     } else {
       printf("Invalid command. Please try again.\n");
     }
  }
}
```

In this banking example, entering a long command causes C to write characters beyond the length of the `command` array. Eventually, it overwrites the memory where `balance` is stored.

This is known as a ***buffer overrun*** (or *buffer overflow*). The C language is especially susceptible to *buffer overruns*, which can cause serious stability and security problems.

In this introductory course, having an array with an appropriate length and using `scanf` is “good enough”.

In practice you would **never** use this insecure method for reading in a string.

If you need to read in a string that includes whitespace until a newline (`\n`) is encountered, the `gets` function can be used (CP:AMA 13.3).

It is also very susceptible to overruns, but is convenient to use in this course.

```
char name[81];
printf("What is your full name?\n");
gets(name);
```

There are C library functions that are more secure than `scanf` and `gets`.

One popular strategy to avoid overruns is to only read in one character at a time (*e.g.,* with `scanf("%c")` or `getchar`). For an example of using `getchar` to avoid overruns, see CP:AMA 13.3.

Two additional <`string.h`> library functions that are useful, but susceptible to buffer overruns are:

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

`strcat(char *dest, const char *src)` copies (appends or con**cat**enates) `src` to the end of `dest`.

You should always ensure that the `dest` array is large enough (and don't forget the null terminator).

Consider this simple implementation of `my_strcpy`:

```
char *my_strcpy(char *dst, const char *src) {
   char *d = dst;
   while (*src) {
      *d = *src;
      ++d; ++src;
   }
   *d = '\0';
   return dst;
}
```

with the following function call:

```
char s[] = "spam";
my_strcpy(s + 4, s);
```

The null terminator of `src` is overwritten, so it will continue to fill up memory with `spamspamspam...` until a crash occurs.

While *writing* to a buffer can cause dangerous buffer overruns, *reading* an improperly terminated string can also cause problems.

```
char c[3] = "cat";    // NOT properly terminated!
printf("%s\n", c);
printf("The length of c is: %d\n", strlen(c));
```

```
cat??????????????????
The length of c is: ??
```

The string library has "safer" versions of many of the functions that stop when a maximum number of characters is reached.

For example, `strnlen`, `strncmp`, `strncpy` and `strncat`.

String literals

The C strings used in statements (*e.g.*, with `printf` and `scanf`) are known as ***string literals***.

```
printf("i = %d\n", i);
printf("the value of j is %d\n", j);
```

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 address of the corresponding array.

The *"read-only"* section is also known as the *"literal pool"*.

example: string literals

```
void foo(int i, int j) {
  printf("i = %d\n", i);
  printf("the value of j is %d\n", j);
}
```

Although no array name is actually given to each literal, it is helpful to imagine that one is:

```
const char foo_string_literal_1[] = "i = %d\n";
const char foo_string_literal_2[] = "the value of j is %d\n";

void foo(int i, int j) {
  printf(foo_string_literal_1, i);
  printf(foo_string_literal_2, j);
}
```

You should not try to modify a string literal. The behaviour is undefined, and it causes an error in `Seashell`.

Arrays vs. pointers

Earlier, we said arrays and pointers are *similar* but **different**.

Consider the following two string definitions:

```
void f(void) {
  char a[] = "pointers are not arrays";
  char *p  = "pointers are not arrays";
  ...
}
```

- The first reserves space for an initialized 24 character array (`a`) in the stack frame (24 bytes).
- The second reserves space for a `char` pointer (`p`) in the stack frame (8 bytes), *initialized* to point at a string literal (`const char` array) created in the read-only data section.

example: more arrays vs. pointers

```
char a[] = "pointers are not arrays";
char *p  = "pointers are not arrays";
char d[] = "different string";
```

`a` is a `char` array. The *identifier* `a` has a constant value (the address of the array), but the elements of `a` can be changed.

```
a = d;            // INVALID
a[0] = 'P';       // VALID
```

`p` is a `char` pointer. `p` is initialized to point at a string literal, but `p` can be changed to point at any `char`.

```
p[0] = 'P';       // INVALID (p points at a const literal)
p = d;            // VALID
p[0] = 'D';       // NOW VALID (p points at d)
```

An array is very similar to a **constant** pointer.

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

In most practical expressions `a` and `p` would be equivalent. The only significant differences between them are:

- `a` is the same as &`a`, while `p` and &`p` have different values
- `sizeof(a)` is 24, while `sizeof(p)` is 8

Goals of this Section

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

- define and initialize arrays and strings
- 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 fixed-length arrays
- describe insertion sort, quicksort and binary search on a sorted array
- represent multi-dimensional data in a single-dimensional array
- explain and demonstrate the use of the null termination convention for strings
- explain string literals and the difference between defining a string array and a string pointer
- sort a string or sequence lexicographically

- use I/O with strings and explain the consequences of buffer overruns
- use <`string.h`> library functions (when provided with a well documented interface)