Arrays & Strings

Readings: CP:AMA 8.1, 8.3, 9.3, 10, 12.1, 12.2, 12.3, 13

Arrays

The only two types of “compound” data storage built-in to C are *structures* and ***arrays***.

Structures can bundle together elements with different *types* and are useful when there are a fixed number of “named” fields.

An *array* is useful when there are an arbitrary number of elements that all have the **same type**.

Arrays in C are used for many tasks where *lists* would be used in Racket, but **arrays and lists are very different**.

To declare an array we must know the length of the array **in advance** (this limitation is discussed in detail later).

Here is an example of an array declaration:

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

To access an element of an array, an ***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]`. The last element is `my_array[5]`.

> Remember, in computer science we always start counting at 0.

An array element can be used just like any other variable. The index does not have to be constant, and can be any expression.

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

int j = a[0];           // j = 4
int k = a[j];           // k = 23
int *p = &a[k-20];      // p points at a[3]

a[2] = a[1] + a[3];
a[0]++;
```

Arrays and **iteration** are a powerful combination:

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

Array efficiency

An attractive feature of arrays is that any element can be accessed in $O(1)$ (constant) time, regardless of the index or the length of the array.

How the $O(1)$ access time is achieved is discussed later.

Racket has a *vector* data type that is very similar to C's arrays:

```
(define v (vector 4 8 15 16 23 42))
```

Like C's arrays, any element of a vector can be accessed by the `vector-ref` function in $O(1)$ time.

Array initialization

An uninitialized array:

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

is zero-filled if the array is *global*. If the array is *local*, it is filled with arbitrary ("garbage") values from the stack.

Like structures, the array initialization syntax uses braces and is only valid when the array is declared.

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
```

The length of the array can be omitted and *automatically* determined from the number of elements in the initialization.

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

Like the braces syntax, omitting the array length only works with initialization.

```
int a[];          // INVALID
a = {1, 2, 3};    // INVALID
```

Similar to structures, C99 added a partial initialization syntax.

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

Omitted elements are initialized to zero.

In C99, the length of the array can be determined *while the program is running* (but you cannot use the initialization syntax).

```
int main(void) {
  int count;
  printf("How many numbers do you need? ");
  scanf("%d",&count);

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

  for (int i=0; i < count; i++) {
    printf("enter #%d: ", i+1);
    scanf("%d", &a[i]);
  }
}
```

Once the *size* of the array is determined, the size of the stack frame is increased.

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.

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

In the above example, the `sizeof(a)` is 24 ($6 \times$ `sizeof(int)`).

Array length

Theoretically, we could use `sizeof` to determine the *length*:

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

In practice, this should never be done, as the `sizeof` operator only properly reports the array size in very specific circumstances.

In C, you must keep track of the array length separately.

Typically, the length is stored in a separate variable (or constant).

Because C does not directly keep track of array lengths, it is a common source of errors and bugs.

C allows access to arrays beyond their length (`a[10000]`) and even allows an expression such as `a[-1]`.

This is one of the common criticisms of C, and most modern languages have fixed this shortcoming.

Passing arrays to functions

Array parameters typically have unknown length (`a[]`), and so the length must be provided as a separate parameter.

```
void print_array(int a[], int len) {
  for (int i = 0; i < len; i++) {
    printf("element %d: %d\n", i, a[i]);
  }
}

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

When calling a function, only the *address* of the array is copied into the stack frame. This is much more efficient than copying the entire array.

However, this allows functions to change (mutate) the array.

It's good style to use the `const` keyword to 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.

Selection sort in C

In our C implementation of *selection sort*, we:

- search for the smallest item remaining in the array
- swap the smallest item to the front of the array
- iterate on the rest of the array

We use a traditional swap helper function.

```
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}
```

```
void selection_sort(int a[], int len) {
  for (int i=0; i < len-1; i++) {
    // find the position of the next smallest
    int minpos = i;
    for (int j = i+1; j < len; j++) {
      if (a[j] < a[minpos]) {
        minpos = j;
      }
    }
    // move (swap) the smallest element to the front
    swap(&a[i], &a[minpos]);
  }
}
```

The running time is $O(n^2)$.

Quick sort in C

In our C implementation of quick sort, we:

- select the last element of the array as our “pivot”
- move all elements that are smaller than the pivot to the front of the array
- move 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 len, int first, int last) {

  if (last <= first) return; // size is <= 1

  int pivot = a[last];       // last element is the pivot
  int pos = first;           // where to put next smallest

  for (int i = first; i < last; i++) {
    if (a[i] <= pivot) {
      swap(&a[pos], &a[i]);
      pos++;
    }
  }
  swap(&a[last], &a[pos]);   // put pivot in correct place
  quick_sort_range(a, len, first, pos-1);
  quick_sort_range(a, len, pos+1, last);
}

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

Binary search

We can write an array `member` function in C that is $O(n)$.

```
bool member(int item, int a[], int len) {
  for (int i=0; i < len; i++) {
    if (a[i] == item) {
      return true;
    }
  }
  return false;
}
```

But what if the array was previously *sorted*?

We can use **binary search** to write an $O(\log n)$ `member` function.

```
bool sorted_member(int item, 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 true;
    } else if (a[mid] < item) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return false;
}
```

The range (`high-low`) starts out at n (`len`) and decreases by $\frac{1}{2}$ each iteration, so the running time is $O(\log n)$.

Pointer arithmetic

Before continuing with arrays, we take a small detour.

When we introduced pointers, we did not discuss any ***pointer arithmetic***.

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

```
int i = 42;
int *p = &i;
int *q = p + 1;

char c = 'a';
char *d = &c;
char *e = d + 1;

address of i  (&i) = 0xf004
value of p     (p) = 0xf004
value of q     (q) = 0xf008

address of c  (&c) = 0xf010
value of d     (d) = 0xf010
value of e     (e) = 0xf011
```

When adding an integer `i` to a pointer `p`, the value of `(p + i)` is equivalent to `(p + i*sizeof(*p))`. Subtracting an integer from a pointer works in the same way.

Pointers can also be incremented (or decremented).

`p++` is equivalent to `p = p + 1`.

It is also possible to subtract a pointer from another pointer. If pointers `p` and `q` are the same type (point to the same type), the value of `(p-q)` is an integer equivalent to `((p-q)/sizeof(*p))`.

In other words, if `q = p + i`, then `i = q - p`.

Pointers can also be compared with the comparison operators:

<, <=, ==, !=, >=, >.

Array pointer notation

In our array example,

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

while the *elements* of `a` are variables, the array identifier `a` is not a "variable" that can be changed (assigned to).

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

While the type of `a` is an array (and *not* a pointer), in practice it behaves like a constant pointer to the start of the array.

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

Strictly speaking, `a` and `p` have different types.

Practically, 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 4
- `p` requires an additional four bytes of storage

Since the array identifier (`a` in this example) is effectively a pointer, `*a` is equivalent to the first element of the array (`a[0]`).

Since we know that array elements are adjacent to each other in memory, `(a + 1)` points at the second element of the array and `*(a + 1)` is equivalent to the second element (`a[1]`).

The square brackets used with arrays (`[]`) are actually an **operator** that performs *pointer arithmetic*.

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

This is why any element can be accessed in $O(1)$ time.

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 (int *p = a; p < a + len; p++) {
    sum += *p;
  }
  return sum;
}
```

The choice of notation (pointers or [ ]) is a matter of style, and it often depends on the context. You are expected to be comfortable with both.

In function *declarations*, `*a` notation is more common than `a[ ]`, even when square brackets are used in the code.

Another example with 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;
  int *p = a;
  while (p < a + len) {
    if (*p == item) {
      count++;
    }
    p++;
  }
  return count;
}
```

The pointer passed to a function does not have to be the first element of the array. The following pointer notation version of `quick_sort` takes advantage of this strategy.

```
void quick_sort(int *start, int len) {
  if (len <= 1) return;

  int *end = start + len - 1;
  int pivot = *end;
  int *pos = start;

  for (int *p = start; p < end; p++) {
    if (*p <= pivot) {
      swap(p,pos);
      pos++;
    }
  }
  swap(pos,end);
  quick_sort(start, pos - start);
  quick_sort(pos + 1, end - pos);
}
```

Multi-dimensional data

All of the arrays seen so far have been one-dimensional arrays.

With one dimensional arrays, we can still represent multi-dimensional data by "mapping" the higher dimensions down to one.

C has native support for multiple-dimension arrays, but they are not covered in this course.

```
int two_dee[10][10];
int three_dee[10][10][10];
```

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

Consider a 9x9 sudoku puzzle ('blanks' are zeros):

```
0,0,0,0,0,0,0,1,4   row 0
2,0,0,5,0,0,6,0,0
9,0,0,3,0,0,0,0,0
...
5,7,0,0,0,0,0,0,0   row 8
```

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

```
int sudoku[81] = {0,0,0,0,0,0,0,1,4,2,0,0,5,0,0,6,0,0,9,...};
```

To access the entry in row `r` and column `c`, we simply access the element at `sudoku[r*9 + c]`.

We can use a similar "mapping" to represent any high-dimensional data with a simple one-dimensional array.

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* (or *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.

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

String initialization

In addition to the regular array initialization syntax, `char` arrays also support a double quote (`"`) *initialization syntax*. When combined with the automatic size syntax (`[]`), the size includes the null terminator.

The following declarations 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 statements (*e.g.* in `printf("string")`).

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* 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"*.

```c
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:

```c
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 RunC.

String declarations: arrays vs. pointers

The following two “string” declarations are **very different**.

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

The first declares a 24 character array `a` on the stack (24 bytes), and *initializes* `a` with the corresponding characters.

The second declares a `char` pointer `p` on the stack (4 bytes), and *initializes* `p` to point at a *string literal* (`const char` array) that is created in the read only section.

```
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 (`const char` array), but `p` can be changed to point at any `char`.

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

Null termination

Because strings are null terminated, we do not have to pass the string length to every function.

```
// char_count(c,s) counts the number of occurrences
//   of the character c within the string s
int char_count(char c, const char *s) {
  int count = 0;
  while (*s != '\0') {          // same as while(*s)
    if (*s == c) count++;
    s++;
  }
  return count;
}
```

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

strlen

The `string` library (`#include <string.h>`) provides many useful functions for processing strings (more on this 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) {
  char *p = s;
  while (*p) {
    p++;
  }
  return (p-s);
}
```

`strlen` is $O(n)$, where n is the length of the string.

Do **NOT** put the `strlen` function within a loop:

```
int char_count(char c, const char *s) {
  int count = 0;
  for (int i=0; i < strlen(s); i++) {    // BAD !!!!
    if (s[i] == c) count++;
  }
  return count;
}
```

By using an $O(n)$ function (`strlen`) inside of the loop, the function becomes $O(n^2)$ instead of $O(n)$.

Unfortunately, this mistake is common amongst beginners.

This will be harshly penalized on assignments & exams.

Lexicographical ordering

When comparing (or sorting) individual characters, the order is determined by the ASCII table.

To compare strings, we use a ***lexicographical order***.

The first characters of each string are compared. If they are different, the comparison ends and the "lower" string is identified. If they are the same (*a tie*), the second characters are compared, and so on. If one of the strings reaches the null terminator, it is the "lower" string. For example:

```
"a" < "c" < "cabin" < "cat" < "catastrophe"
```

The <string.h> library function strcmp(a,b) uses lexicographical ordering, and returns zero if the strings a and b are identical. If a is "lower" than b, it returns a negative integer, otherwise it returns a positive integer.

```
int my_strcmp(const char *a, const char *b) {
  while (*a == *b) {
    if (*a == '\0') return 0;
    a++;
    b++;
  }
  if (*a < *b) return -1;
  return 1;
}
```

strcmp is $O(n)$, where n is the smallest length of strings a and b.

To compare if two strings `a` and `b` are *equal*, we must use the `strcmp` function.

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

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

if (a == b) {...}                  // NEVER TRUE
if (strcmp(a,b) == 0) { ... }      // PROPER COMPARISON
```

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

For example, 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);
```

With `%s`, `printf` prints out characters until the null character is encountered. The running time of `printf` is $O(n)$ where n is the length of the string.

Because string literals have constant length, printing individual values is still considered $O(1)$.

```
printf("long string literals are size O(%d)\n", 1);
```

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

`scanf` is useful for reading in one "word" at a time.

```
char name[81];
printf("What is your first name? ");
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?

```c
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 size of the `command` array. Eventually, it will overwrite 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 appropriately sized array and using `scanf` is "good enough".

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

The `gets` function does not stop when a space is encountered, and reads in characters until a newline (`\n`) is encountered. It is also very susceptible to overruns, but is convenient to use in this course.

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

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

A popular strategy to avoid overruns is to only read in one character at a time (*e.g.*, with `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).

You can crash your program with a simple call to `strcpy`:

```
char c[] = "spam";
strcpy(c + 4, c);
```

Because the start of the destination is also the null terminator of the source, the source will never terminate and it will fill up the 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`.

Goals of this module

You should be comfortable declaring and initializing arrays and strings.

You should be comfortable using iteration to loop through arrays.

You should be comfortable using pointer arithmetic.

You should understand how arrays are represented in the memory model, and how the array index operator (`[ ]`) uses pointer arithmetic to access array elements in $O(1)$ time.

You should be comfortable with both array index notation (`[ ]`) and array pointer notation and converting between the two.

You should understand the array implementations of selection sort, quick sort and binary search.

You should be comfortable representing multi-dimensional data in a single-dimensional array.

You should understand string literals and the difference between declaring a string array and a string pointer.

You should understand the null termination convention and the importance of properly terminating strings.

You should understand lexicographical ordering, and be able to sort a sequence lexicographically.

You should be comfortable using I/O with strings and understand the consequences of buffer overruns.

You should be comfortable using <string.h> library functions (if provided with a well documented interface).