Chapter 8 -- data structures

```
DATA STRUCTURES
---------------

common theme in programming:
  SPACE vs. TIME tradeoff
       space is memory space
       time is time to execute program

  it is often possible to write a program such that it
    1. executes very fast, but wastes/utilizes more memory
     or
    2. utilizes little memory, but executes quite slow

  data structures can make memory useage efficient or inefficient


data structures we will discuss:
  arrays, stacks, and queues  (in that order)

ARRAYS
------

array implementation is important
 1. most assembly languages have no concept of arrays
 2. from an array, any other data structure we might want can be built

Properties of arrays:
 1. each element is the same size (char = 1 byte, integer = 4 bytes)
 2. elements are stored contiguously, with the first element
    stored at the smallest memory address

so, the whole trick in assembly language is
  1.  allocate correct amount of space for an array
  2.  an ADDRESS tells the location of an element
  3.  USE address to get at array element




memory can be thought of as an array
    it is a giant array of bits or bytes or words (picture needed here!)
    the element numbering starts at 0
    the element number is an address

    if each byte has a unique address, we have BYTE ADDRESSING.
    (the Pentium has this, as do many machines)

    if each doubleword has a unique address, we have WORD ADDRESSING.



SASM access of the memory array
```

we can access any element (byte) of memory

m(address) refers to the element (contents) at address
m(20) is the byte or doubleword numbered address 20,
the 21st byte of memory.

important: 20 is the address
m(20) is the contents at address 20

SASM declarations of arrays within memory

to allocate a portion of memory (more than a single variable's worth)
(allocating an array within memory)

variablename type numelements dup(initvalue)

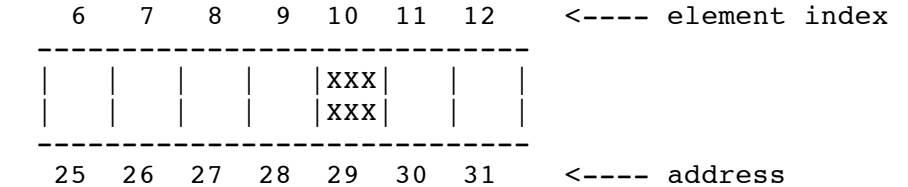
type is just like before -- db or dd
numelements is just that, and numbering ALWAYS starts at 0
initvalue is a value given to each element of the array

example:

my_array db 8 dup(0)
8 character elements, numbered 0 - 7, initialized to 0

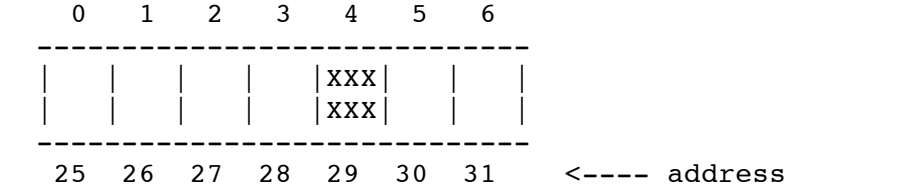
an example of how to calculate the address of an element:

byte (character) elements --
array1: array[6..12] of char; /* PASCAL */



byte address of array1[10] = 25 + (10 - 6)
= 29

this same example (or close) only in SASM:
array1 db 7 dup(0)



```
       want the 5th element,

            array1[4] is at address     array1 + 4
            If element[0] is at address 25,
         byte address of array1[4] =   25 + 4
```

how do you get the address array1?

SASM la (load address) instruction

```
la   ar_addr, array1
```

takes the address array1 (25 in the example) and puts it into the variable called ar_addr.

this is where it is extremely important to understand and keep clear the difference between an address and the contents of an address.

to reference array1[4] in SASM, write the code,

```
   la  my_addr, array1
   iadd my_addr, 4

   ; then if  we wanted to decrement element number 4
   sub m(my_addr), 1
```

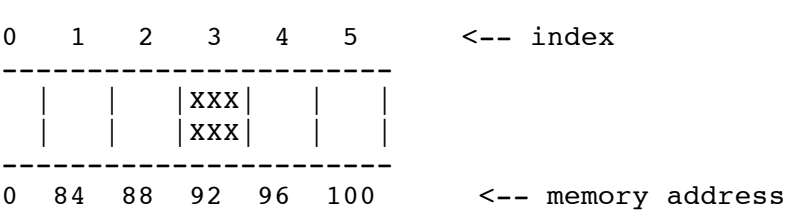
remember, my_addr is an address (declared as dd)
m(my_addr) is the byte at the address my_addr

integer elements --

```
array2:  array[0..5] of integer;  /* PASCAL */
```

in SASM,

```
array2  dd  6 dup(0)
```




```
byte address of array2[3] =  80 + 4(3 - 0)
                          =  92
```

SO, we need to know

1. where the array starts (called base address)
2. size of an element in bytes (to get a byte address)
3. what the first element is numbered

byte address of element[x] = base + size(x - first index)

Note: if we always start the index numbering at 0, then

this formula becomes

byte address of element[x] = base + size(x)

2 DIMENSIONAL ARRAYS

There are more issues here, than for 1 dimensional arrays.

First, how to map a 2 dimensional array onto a 1 dimensional memory?

TERMINOLOGY:

r x c array -- r rows
c columns

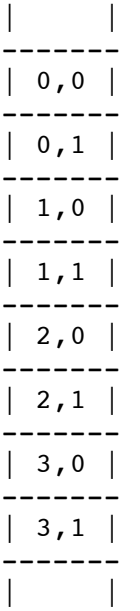
element[y, x] -- y is row number
x is column number

example: 4 x 2 array

mapping this 4 x 2 array into memory.
2 possiblilities:

row major order:

rows are all together



column major order:

columns are all together

```
-------
| 0,0 |
-------
| 1,0 |
-------
| 2,0 |
-------
| 3,0 |
-------
| 0,1 |
-------
| 1,1 |
-------
| 2,1 |
-------
| 3,1 |
-------
|     |
```

2-D Arrays

The goal: come up with a formula for calculating the address of an element of a 2-D array.

Row Major:

addr. of [y, x] = base + offset to correct row + offset within row

(size)(y - first_row) (# columns)

(size) (x - first_col)

Column Major:

addr. of [y, x] = base + offset to correct column + offset within column

(size)(x - first_col) (# rows)

(size) (y - first_row)

Need to know:

1. row/column major (storage order)
2. base address
3. size of elements

4. dimensions of the array

HINTS toward getting this correct:
Draw pictures.
Don't forget to account for size.

BOUNDS CHECKING

Many HLL's offer some form of bounds checking. Your program crashes, or you get an error message if an array index is out of bounds

```
   cal example:      x:  array[1..6] of integer;
                code. . .

                    y := x[8];
```

Assembly languages offer no implied bounds checking.
After all, if your program calculates an address of an element, and then loads that element (by the use of the address), there is no checking to see that the address calculated was actually within the array!

example (to motivate some thought as to how to do bounds checking):

```
   given --      a 5 x 3 array
                 byte size elements
                 row major order
                 first_row = 1
                 first_col = 1

         what is the address of element[2, 5]
```

a program probably just plugs the numbers into the formula:

```
addr of [2, 5] = base + 1(2 - 1)(3) + 1(5)
               = base + 8
```

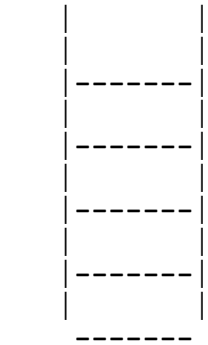
this actually gives the address of element [3, 3], still a valid element of the array, but not what was really required. . .

STACKS

We often need a data structure that stores data in the reverse order that it is used. Along with this is the concept that the data is not known until the program is executed (RUN TIME). A STACK allows both properties.

Abstractly, here is a stack. Analogy to stack of dishes.

Also dubbed Last In First Out, LIFO.



Data put into the stack is said to be PUSHED onto the stack.
Data taken out of the stack is said to be POPPED off the stack.

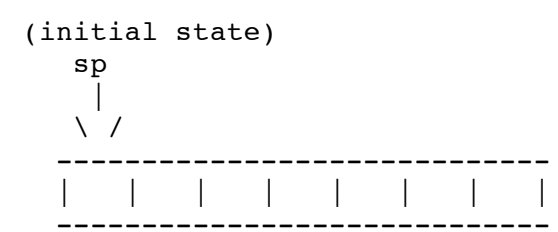
Algorithm Example:
printing out a positive integer, character by character
(integer to character string conversion)

```
integer = 1024

if integer == 0 then
   push '0'
else
   while integer <> 0
      digit <- integer mod base
      char <- digit + 48
      push char onto stack
      integer <- integer div base

while stack is not empty
   pop char
   put char
```

IMPLEMENTATION OF A STACK

One implementation of a stack out of an array.

Need to know:
index of TOP OF STACK (tos), often called a stack pointer (sp).
In most implementations, tos is not an index at all, it is
an ADDRESS (pointer).



sp is a variable that contains the address of the empty location
at the top of the stack.

for an array declared (in SASM) as

```
       my_stack  dd  50 dup(0)
       my_sp     dd  ?
```

OR (could be, since amount of space is important)

```
       my_stack  db  200
       my_sp     dd  ?  ; doesn't change in these declarations
```

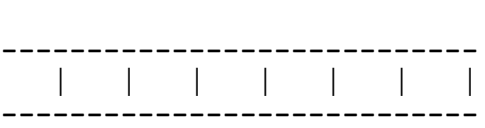
a PUSH operation:

```
      move    m(my_sp), data
      iadd    my_sp, 4
```

a POP operation:

```
      isub    my_sp, 4
      move    data, M(my_sp)
```

A stack could instead be implemented such that the stack pointer points to a FULL location at the top of the stack.



a PUSH operation:

```
      iadd    my_sp, 4
      move    M(my_sp), data
```

a POP operation:

```
      move    data, M(my_sp)
      isub    my_sp, 4
```

ANOTHER ALTERNATIVE:
The stack could "grow" from the end of the array towards the beginning. (Note that which end of the array the stack grows toward is independent of what stack pointer points to.)

For the student to figure out:
How do you initialize the stack pointer?
How do you know when there are no more items in the stack?
How do you know when the stack is full?

```
QUEUES
-------

whereas a stack is LIFO,
        a queue is FIFO (First In, First Out).

a real life analogy is a line (called a queue in British English).
  Person gets on the end of the line (the TAIL),
    waits,
      and gets off at the front of the line (the HEAD).

getting into the queue is an operation called ENQUEUE.
taking something off the queue is an operation called DEQUEUE.

it takes 2 pointers to keep track of the data structure,
  the head and the tail.


  initial state:
  ---------------------------------------------
      |     |     |     |     |     |
  ---------------------------------------------
              ^
              |
              head, and tail


  after 1 enqueue operation:
  ---------------------------------------------
      |     |  x  |     |     |     |
  ---------------------------------------------
               ^     ^
               |     |
               |     tail
               head

  after another enqueue operation:
  ---------------------------------------------
      |     |  x  |  y  |     |     |
  ---------------------------------------------
               ^           ^
               |           |
               |           tail
               head

  after a dequeue operation:
  ---------------------------------------------
      |     |  x  |  y  |     |     |
  ---------------------------------------------
                     ^     ^
                     |     |
                     |     tail
                    head



Note that (like stacks) when an item is removed from
the data structure, it is physically still present,
```

but correct use of the structure cannot access it.

If enough items are enqueued (and possibly dequeued) from the queue, the points will eventually run off the end of the array! This leads to implementations that "wrap" the beginning of the array to the end, and forms a CIRCULAR QUEUE.

The implementation of the circular queue is a bit more complex. The conditions to test for an empty queue and full queue are more difficult. They can be eased by implementing a queue with one element that is a DUMMY. It is never used for data storage. The DUMMY element works its way around the memory allocated for the queue as items are enqueued and dequeued.

This is an example of the space vs. time trade-off. An extra piece of memory is used in an inefficient manner, in order to make the test for full/empty queues more efficient.