



## Pointer Arithmetic

- Address of element  $i$  of array  $x$  is equal to:  
*address of element 0 of array  $x$  + size in bytes of an array element  $\times i$*
- Example in C:
  - `int x[20];`
  - Value `x[12] = *(address of byte at x[0] + 12 * sizeof(int))`
  - In code: `*(&x[0] + 12)`
    - Note that C multiplies 12 by `sizeof(int)` implicitly
- We have done things like this in assembly already

## A Full Example

- Let's write a function to count the number of occurrences of each capital letter in a null-terminated arbitrary string
  - This is very much a "tally computation" problem
- This function would have (in C) the following prototype:  
`void print_tallies(char *array)`
- The function needs a local array of tallies (with 26 elements)
  - Let's say that each tally will be stored as a 2-byte number
- So we have a function that takes an array as an argument (no need to pass its size as well since it is null-terminated) and uses an array as a local variable
- Let's see how to write this code

## A Full Example

```

#include "asm_io.inc"

segment .data
    string db "I REALLY HATE WRITING ASSEMBLY CODE", 0
    msg db ":", 0
segment .bss

segment .text
    global asm_main
asm_main:
    enter 0,0 ; setup
    pusha ; setup

    ;; Main program ;;
    push string ; push the argument to print_tallies
    call print_tallies ; call print_tallies
    add esp, 4 ; remove the argument from the stack
    jmp end ; go to the end of the program

    ;; print_tallies function here

end:
    popa ; cleanup
    mov eax, 0 ; cleanup
    leave ; cleanup
    ret ; cleanup
    
```

## The print\_tallies function

```

    ;; print_tallies function ;;
    ;; arguments: 1 4-byte address
    ;; local variables: a 26-element single word array (52 bytes)

print_tallies:
    push ebp ; save ebp
    mov ebp, esp ; set ebp to esp
    sub esp, 52 ; add stack space for 52 bytes!

    ;; body of the function here

    mov esp, ebp ; clean up the stack
    pop ebp ; restore ebp
    ret ; return
    
```

EBP+8	@ of the string
EBP+4	return address
EBP	saved EBP
26 2-byte elements	
EBP - 50	EBP - 52

## The print\_tallies function

- Initialize the array of tallies

```

    ;; initialize tally array
    mov ecx, ebp ; set ecx to ebp
    sub ecx, 52 ; set ecx to the @ of the first tally

init_loop:
    mov word [ecx], 0 ; set the current tally to 0
    add ecx, 2 ; ecx += 2 (move on to the next tally)
    cmp ecx, ebp ; if reached the end of the tallies, stop
    jnz init_loop ; otherwise keep looping
    
```

## The print\_tallies function

- Compute the tallies

```

    mov ebx, [ebp+8] ; set ebx to the first character of the string argument

tally_loop:
    cmp byte [ebx], 0 ; is the current byte null?
    jz end_tally_loop ; if so, exit the loop
    mov al, [ebx] ; load the byte into al
    sub al, 65 ; subtract the ASCII code for 'A'
    mov ecx, ebp ; set ecx to ebp
    sub ecx, 52 ; set ecx to the address of the first tally
    shl al, 1 ; multiply al by two
    movzx eax, al ; zero extend al into eax
    add ecx, eax ; add eax to ecx, so that ecx points to the right tally
    inc dword [ecx] ; increment the tally
    inc ebx ; increment ebx to point to the next byte
    jmp tally_loop ; loop

end_tally_loop:
    
```

## The print\_tallies function

- Print the tallies

```

mov     ecx, ebp           ; set ecx to ebp
sub     ecx, 52           ; set ecx to the address of the first tally
mov     bl, 65            ; set bl to the ASCII code for 'A'
print_tally_loop:
movzx   eax, bl           ; print the current character
call    print_char       ; print the current character
mov     eax, msg          ; print ". "
call    print_string     ; print ". "
mov     ax, [ecx]         ; ax = current tally
movzx   eax, ax           ; zero extend the tally
call    print_int        ; print the tally
call    print_nl         ; print a new line
inc     bl                ; increment bl
add     ecx, 2            ; ecx points to the next tally
cmp     ecx, ebp         ; if not beyond the last tally
jnz     print_tally_loop ; keep looping

```

## Multi-Dimensional Arrays

- We, as programmers, like to think of multi-dimensional arrays as rectangles (2D), or cubes (3D)
  - Pretty hard to visualize the geometry of D-dimensional objects when  $D > 3$
- The computer stores all multi-dimensional arrays as 1-D arrays
- Multi-dimensional arrays are just “chopped” into pieces and fit into 1-D arrays in memory
  - After all, the computer’s memory is 1-D, not multi-D
  - It’s just a sequence of byte addresses!

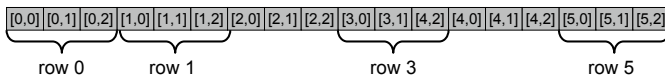
## 2-D Arrays

int x[6][4];

row-major

[0,0]	[0,1]	[0,2]
[1,0]	[1,1]	[1,2]
[2,0]	[2,1]	[2,2]
[3,0]	[3,1]	[4,2]
[4,0]	[4,1]	[4,2]
[5,0]	[5,1]	[5,2]

in memory:



$$\text{address of } x[i,j] = \text{address of } x[0,0] + i * \text{number of columns} * \text{sizeof(int)} + j * \text{sizeof(int)}$$

## In-class Exercise

- Consider the following declaration: `int x[12][8];`
- Assume that the 2-byte address of `x[0,0]` is `004Dh`
- What’s the address of `x[3,6]`?

- Consider the following declaration: `char y[32][32];`
- Assume that the 2-byte address of `y[0,0]` is `0400h`
- What’s the address of `y[10,2]`?

## In-class Exercise

- Consider the following declaration: `int x[12][8];`
- Assume that the 2-byte address of `x[0,0]` is `004Dh`
- What’s the address of `x[3,6]`?
- address of `x[3,6]` =  $004Dh + (3 * 8 * 4)d + (6 * 4)d$ 
  - =  $004Dh + 96d + 24d$
  - =  $004Dh + 60h + 18h$
  - =  $004Dh + 0078h$
  - =  $00C5h$

## In-class Exercise

- Consider the following declaration: `char y[32][32];`
- Assume that the 2-byte address of `y[0,0]` is `0400h`
- What’s the address of `y[10,2]`?
- Address of `y[10,2]` =  $0400h + (32 * 1 * 10)d + (2 * 1)d$ 
  - =  $0400h + 322d$
  - =  $0400h + 0142h$
  - =  $0542h$

## 3-D Arrays?

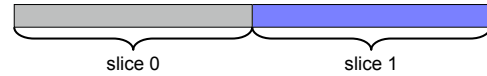
- We “chopped” a 2-D array into a bunch of 1-D arrays (rows) to fit it into 1-D memory
- Similarly, we “chop” a 3-D array into a bunch of 2-D arrays (slices), that we know how to fit into 1-D memory
- More generally: we consider an n-D array as a bunch of (n-1)-D arrays, and we can thus store it in memory recursively
- Let’s see 3-D array example

## 3-D Arrays

```
int mat[m,n,p];  
int mat[2,5,4];
```

[0,0,0]	[0,0,1]	[0,0,2]	[0,0,3]	[1,0,0]	[1,0,1]	[1,0,2]	[1,0,3]
[0,1,0]	[0,1,1]	[0,1,2]	[0,1,3]	[1,1,0]	[1,1,1]	[1,1,2]	[1,1,3]
[0,2,0]	[0,2,1]	[0,2,2]	[0,2,3]	[1,2,0]	[1,2,1]	[1,2,2]	[1,2,3]
[0,3,0]	[0,3,1]	[0,3,2]	[0,3,3]	[1,3,0]	[1,3,1]	[1,3,2]	[1,3,3]
[0,4,0]	[0,4,1]	[0,4,2]	[0,4,3]	[1,4,0]	[1,4,1]	[1,4,2]	[1,4,3]

in memory:



$$\begin{aligned} \text{address of } x[i,j,k] &= \text{address of } x[0,0,0] + \\ & i * (n * p) * \text{sizeof(int)} + \\ & j * p * \text{sizeof(int)} + \\ & k * \text{sizeof(int)} \end{aligned}$$

## Conclusions

- High-level languages provide an “array abstraction”
  - Makes life easy and can allow us to think of arrays as multi-dimensional geometric objects
  - But internally everything’s 1-D
- The abstraction hides a lot of work
  - Example: calculating the address of  $a[i,j,k]$  requires 3 additions and 6 multiplications!
- In assembly we do not have such abstraction
  - more work for us
  - But perhaps opportunities to optimize element access