

Arrays

CS31

Pascal Van Hentenryck



Overview

Arrays in assembly language

- 1 Dimensional
- 2 Dimensional

The Easy Case

One-dimensional array

- Elements of size 1
- Indexing starts at 0

```
stuff:      .byte      0:10
```

Addresses of elements:

- stuff
- stuff + 1
- stuff + 2
- ...

So what is the address of stuff [i] ?

stuff + i

What happens if 'i' is bigger than the number of elements you've declared in the array?

- You get something not in your array
- Or... you get a non-valid address

Loading

To get element i of `stuff` into register `$s1`:

```
li    $s0, i
lb    $s1, stuff($s0)
```

If you really knew $i=3$ in advance, you could say:

```
lb    $s1, stuff+3
```

Fancier Indexing

In Pascal, we might have

```
stuff: array[-4..4] of byte;
```

Or, more generally,

```
stuff: array[xfirst..xlast] of byte;
```

Declaring the array:

```
      xfirst = -4  
      xlast  = 4  
      xnb    = 9  
stuff:      .byte    0:xnb
```

More Fancy Indexing

Where is `stuff[-4]`?

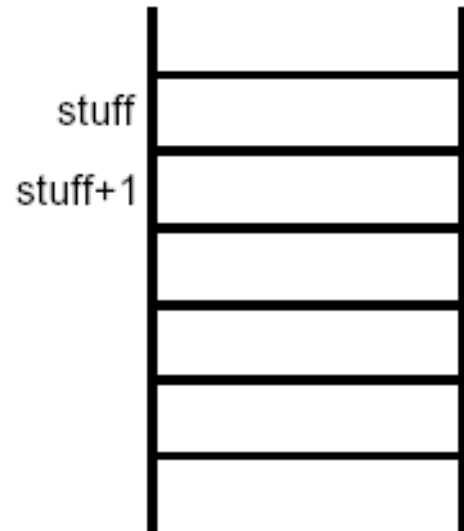
$$\text{stuff} + -4 - (-4) = \text{stuff}$$

Where is `stuff[0]`?

$$\text{stuff} + 0 - (-4) = \text{stuff} -$$

Where is `stuff[4]`?

$$\text{stuff} + 4 - (-4) = \text{stuff} + 8$$



More Fancy Indexing

Where is `stuff[-4]`?

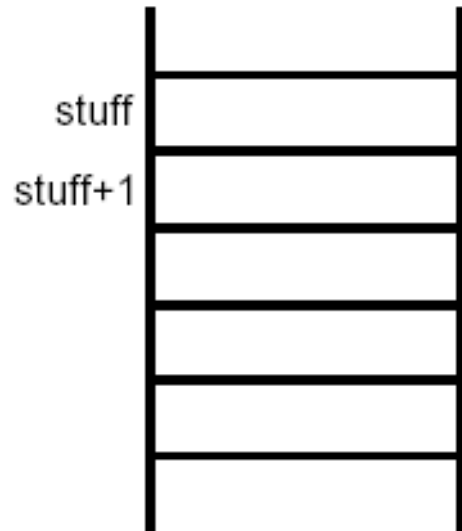
$$\text{stuff} + -4 - (-4) = \text{stuff}$$

Where is `stuff[0]`?

$$\text{stuff} + 0 - (-4) = \text{stuff} + 4$$

Where is `stuff[4]`?

$$\text{stuff} + 4 - (-4) = \text{stuff} + 8$$



In general, `stuff[i]` is at address:

$$\text{stuff} + i - \text{xfirst}$$

To load:

```
li    $s0, i
sub   $s0, $s0, xfirst
lb    $s1, stuff($s0)
```

Bigger Things

What if we have

```
xfirst = 0
xlast = 9
stuff:.word    0:10
```

Where is `stuff[0]` ?

`stuff + 0`

Where is `stuff[5]` ?

`stuff + 4*5`

In general, `stuff[i]` is at address:

`stuff + 4*i`

Much Bigger Things

```
xfirst = 3
xlast = 9
xsize = 158           #size in bytes
stuff: .byte 0:xnb
```

Where is `stuff[i]` ?

`stuff + (i - xfirst) * size`

To get the first word of *i*'th element:

```
li    $s0, i
sub   $s0, $s0, xfirst
mul   $s0, $s0, xsize
lb    $s1, stuff($s0)
```

Clearing a 1-D Array

```
xfirst = 2
xlast = 8
xsize = 4
```

```
.data
stuff: .word 0:28
endstuff:
```

```
#clear the array
#$s0: address
```

```
.text
__start:   la    $s0,stuff
           la    $s1,endstuff
loop:     beq   $s0,$s1,finished
           sw    $0,($s0)
           add   $s0,$s0,xsize
           j    loop
finished: done
```

Two-Dimensional Array

In Pascal:

```
stuff: array[0..2,0..4] of byte;
```

Row major order:

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

Another view:

row \ col	0	1	2	3	4
0	stuff+0	stuff+1	stuff+2	stuff+3	stuff+4
1	stuff+5	stuff+6	stuff+7	stuff+8	stuff+9
2	stuff+10	stuff+11	stuff+12	stuff+13	stuff+14

More Two Dimensions

```
stuff: array[0..2,0..4] of byte;
```

Where is `stuff[0,0]`?

`stuff`

Where is `stuff[0,2]`?

`stuff + 2`

Where is `stuff[2,0]`?

`stuff + 2*5`

Where is `stuff[2,3]`?

`stuff + 2*5 + 3`

Declaring the array

```
rowfirst = 0
```

```
rowlast  = 2
```

```
colfirst = 0
```

```
collast  = 4
```

```
.data
```

```
stuff:   .byte      0:15
```

size is $(\text{rowlast} - \text{rowfirst} + 1)(\text{collast} - \text{colfirst} + 1)$

Arbitrary Indices

`stuff: array[6..10,3..5] of byte;`

Stuff:

s[6,3]	s[6,4]	s[6,5]	s[7,3]
s[7,4]	s[7,5]	s[8,3]	s[8,4]
s[8,5]	s[9,3]	s[9,4]	s[9,5]
s[10,3]	s[10,4]	s[10,5]	

In general, `stuff [i,j]` is at:

$\text{stuff} + (i - \text{rowfirst}) * (\text{collast} - \text{colfirst} + 1) + (j - \text{colfirst})$

(More) Bigger Things

```
rowfirst = 6
rowlast  = 10
colfirst  = 3
collast   = 5
size      = 4           #size in bytes
```

In general, `stuff [i,j]` is at:

`stuff+size[(i-rowfirst)*(collast-colfirst+1)+(j-colfirst)]`

row \ col	3	4	5
6	stuff+0	stuff+4	stuff+8
7	stuff+12	stuff+16	stuff+20
8	stuff+24	stuff+28	stuff+32
9	stuff+36	stuff+40	stuff+44
10	stuff+48	stuff+52	stuff+56

Making it Efficient

let rsize = collast - colfirst + 1

stuff[i,j] is at:

```
stuff+size*[i*rsize-rowfirst*rsize+j-colfirst]  
=stuff+size*[i*rsize+j-(rowfirst*rsize+colfirst)]
```

2-D Example

```
# Puts value 0xff along the  
main diagonal of a square  
array
```

```
rowfirst      = 6  
rowlast      = 9  
colfirst     = 3  
collast      = 6  
size         = 4  
value        = 0xff
```

```
stuff:        .data  
              .word      0:16
```

```

.text
__start:
# $t0: the value we will put into array
# $t1: the number of columns
# $s0: offset into array
# $s1: current row index
# $s2: current column index

        li        $t0, value
        li        $s1, rowfirst
        li        $s2, colfirst
        li        $s3, rowlast      # doesn't change
        li        $s4, collast      # doesn't change
        sub       $t1, $s4, colfirst

loop:   bgt       $s1, $s3, finished

# address calculation
        sub       $t2, $s1, rowfirst # i - rowfirst
        mul       $s0, $t1, $t2      # * nbc
        add       $s0, $s0, $s2      # + col
        sub       $s0, $s0, colfirst # - colfirst
        mul       $s0, $s0, size     # * size

# end address calculation
        sw        $t0, stuff($s0)
        add       $s1, $s1, 1
        add       $s2, $s2, 1
        j         loop

finished:

```

Passing Parameters by Reference

In the previous examples, we have passed actual values as parameters.

What if we want to pass an *array* as a parameter.

Rather than stuffing all those values on the stack, we can simply pass its *address* (the base address) as a parameter.

This is known as passing parameters *by reference*.

Passing Arrays to Subroutine

How do we pass array as parameter?

Passing by value

- Put the size and all values on the stack.

Passing by *reference*.

- Pass the address of the array

Array as a Parameter

Here's a subroutine that fills a one-dimensional array of words with a given value. Parameters are

```
addr:    address
len:     number of elements in the array
value:   value to fill in array elements
```

and will be passed on the stack in that order.

```
        # t0: pointer incremented thru array
        # t1: address of end of array
        # t2: value
fill:   sub     $sp,$sp,4           # push ra
        sw     $ra,4($sp)
        lw     $t0,16($sp)        # get start array
        lw     $t1,12($sp)        # get len. (elts)
        mul    $t1,$t1,4          # len. in bytes
        add    $t1,$t1,$t0        # end addr.
loop:   lw     $t2,8($sp)         # get value
        bgt   $t0,$t1,done
        sw    $t2,($t0)          # asgn val to elt
        add   $t0,$t0,4          # incr. pointer
        j    loop
done:   lw     $ra,4($sp)
        add   $sp,$sp,4
        jr    $ra
```