

DATA STRUCTURES (1)

- **Array**: A set of data blocks (bytes, words, arrays, ...) of *equal size, stored contiguously* in memory
 - ▷ Memory is an array of bytes or words (depending upon access mode)
 - In SAL:
 - ◇ To refer to the byte located at **address**, use **m[address]**
 - ◇ To refer to the word starting at **address**, use **M[address]**
(where **address** is a multiple of 4)
 - ▷ The general-purpose registers are an array (the **register file**)
 - Registers can be addressed by number (\$0–\$31); for example,
add \$8, \$9, \$10
 - ▷ The floating-point registers are also an array
 - Must be addressed in pairs (\$f0, \$f2, \$f4, ... , \$f30)

DATA STRUCTURES (2)

- Arrays are built into higher-level languages
 - ▷ In FORTRAN:
 - To declare a one-dimensional array:
`real*4 f(2000)`
 - To store a value in an array element:
`f(i) = 1.e-3 * float(i)`
 - ▷ In C:
 - To declare a one-dimensional array:
`float f[2000];`
 - To declare a pointer array:
`float *fp[2000];`
 - To store a value in an array element:
`f[i] = 1.e-3 * ((float) i);`

DATA STRUCTURES (3)

- Uses of arrays in EE & physics
 - ▷ **Sampled values of a function**

- One dimension:

$$f1[i] = f_1(x_i)$$

- Two dimensions:

$$f2[i][j] = f_2(x_i, y_j) \quad \text{or matrix element } f_{i,j}$$

- Three dimensions:

$$f3[i][j][k] = f_3(x_i, y_j, z_k)$$

- Four dimensions:

$$f4[i][j][k][l] = f_4(x_i, y_j, z_k, t_l)$$

- ▷ Strings
- ▷ Pointer arrays

DATA STRUCTURES (4)

- Assemblers don't know about arrays
 - ▷ Reserve space for an array of 2000 integers in SAL/SPIM:
`ar: .word 0:2000`
 - ▷ Load starting address of array `ar` into register `$v0`:
`la $v0, ar`
 - ▷ To access an array element, must give its address as **base + offset**
 - Computation of the address of `ar[i]` by the C compiler:
`address of ar[i] = address of ar[0] + sizeof(ar[0]) * i`
 - Computation of the address of `ar(i)` by the FORTRAN compiler:
`address of ar(i) = address of ar(1) + sizeof(ar(1)) * (i-1)`
 - `sizeof(f(1))` = number of bytes occupied by the element `ar(1)`

DATA STRUCTURES (5)

- Example of storing a number in an array element in SAL:

```
faddr: .word          # pointer to f[0]
i:     .word          # array index
iaddr: .word          # pointer to f[i]
offset: .word         # offset of f[i] from f[0], in bytes
f:     .word 0:80     # 80 = no. of array elements
step:  .word 5
x:     .word
      :
mul    x, step, i     # x = step * i;
la     faddr, f       # get address of f[0]
mul    offset, i, 4   # compute offset of f[i] IN WORDS!!!!
add    iaddr, offset, faddr # compute absolute address of f[i]
move   M[iaddr], x   # f[i] = step * i;
```

```

# SPIM code to initialize the elements of a 1-d array
# with the value stored in location val

.data
ar1:      .word    0:20  # array of 20 integers (4 bytes each)
val:      .float   1.e-3 # value to store in array elements
nelems:   .word    20   # no. of array elements - 1; max. value
                        # of array index
size:     .word    4    # size of an array element, in bytes

# Register usage
# For clarity, each register holds only one variable
# Nobody would program in this way, because some registers can be re-used
#
# $t0      base address (address of ar1[0])
# $t1      size = size of an array element (in bytes)
# $t2      nelems = number of array elements
# $t3      nmax = nelems - 1 = max. value of array index
# $t4      index = array index of current element
# $t5      addr = absolute address of ar1[index]
# $t6      offset = index * size
# $t7      n = counter to be decremented
# $t8      val = value to be stored in each array element

.text
__start:
    la  $t0,ar1      # get absolute address of ar1[0]
                        # addresses of other array elements
                        # will be computed as base + offset

    lw  $t1,size
    lw  $t2,nelems
    addi $t3,$t2,-1   # calculate nmax = nelems - 1
    ori  $t4, $0, 0   # initialize index to 0
    lwcl $f0, val
    mfcl $t8, $0
loop:   mul    $t6,$t4,$t1 # loop begins here; offset in bytes
        add    $t5,$t6,$t0 # compute absolute address of ar1[index]
        sw    $t8,0($t5)  # store val in ar1[index]
        addi  $t4,$t4,1   # increment array index

```

ASSEMBLED TEXT FOR init1d.s

```

[0x00400000] 0x3c081001 lui $8, 4097 # la $t0,ar1 # get absolute
[0x00400004] 0x3c011001 lui $1, 4097
[0x00400008] 0x8c290068 lw $9, 104($1) # lw $t1,offset
[0x0040000c] 0x3c011001 lui $1, 4097
[0x00400010] 0x8c2a0058 lw $10, 88($1) # lw $t2,index
[0x00400014] 0x3c011001 lui $1, 4097
[0x00400018] 0x8c2b0060 lw $11, 96($1) # lw $t3,size
[0x0040001c] 0x3c011001 lui $1, 4097
[0x00400020] 0x8c2c0054 lw $12, 84($1) # lw $t4,base
[0x00400024] 0x3c011001 lui $1, 4097
[0x00400028] 0x8c2e0050 lw $14, 80($1) # lw $t6,val
[0x0040002c] 0x3c011001 lui $1, 4097
[0x00400030] 0x8c2f005c lw $15, 92($1) # lw $t7,nmax
[0x00400034] 0x014b0018 mult $10, $11
[0x00400038] 0x00004812 mflo $9 # mul $t1,$t2,$t3 # loop begins
[0x0040003c] 0x012c6820 add $13, $9, $12 # add $t5,$t1,$t4 # co
[0x00400040] 0xadae0000 sw $14, 0($13) # sw $t6,0($t5) # store val
[0x00400044] 0x214a0001 addi $10, $10, 1 # addi $t2,$t2,1 # increme
[0x00400048] 0x01eac022 sub $24, $15, $10 # sub $t8,$t7,$t2 # subt
[0x0040004c] 0x0701fffa bgez $24 -24 # bgez $t8, loop # branch back
[0x00400050] 0x3402000a ori $2, $0, 10 # ori $v0,$0,10
[0x00400054] 0x0000000c syscall # syscall

```

DATA STRUCTURES (6)

● Two-dimensional arrays

▷ In C:

○ Row-major storage order

○ address of `ar2[i][j]`

$$= \text{ar2} + \text{data_size} * \text{no_columns} * i + \text{data_size} * j$$

where `ar2` = address of `ar2[0][0]`

▷ In FORTRAN:

○ Column-major storage order

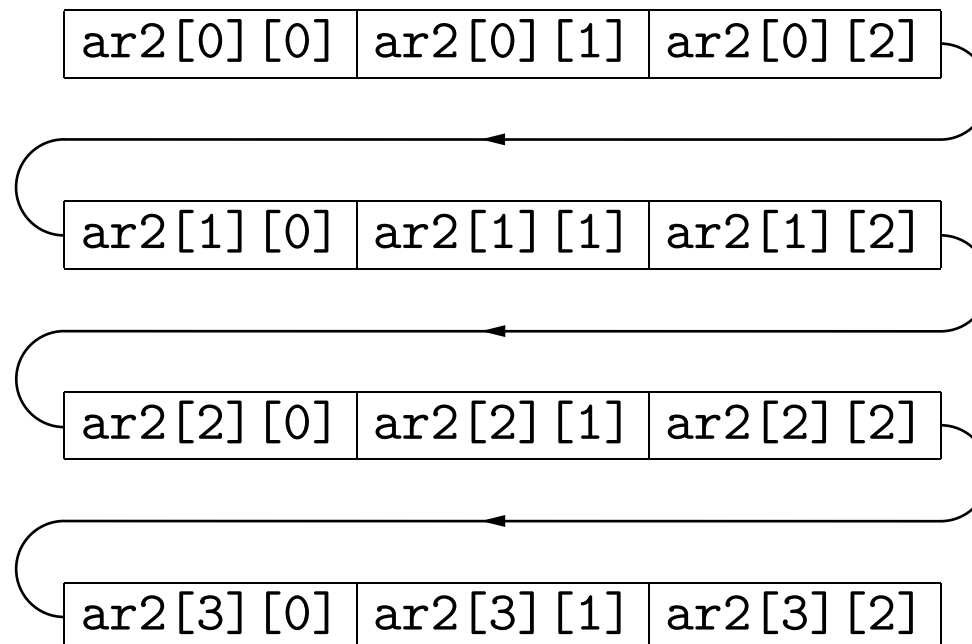
○ address of `ar2(i,j)`

$$= \text{ar2} + \text{data_size} * \text{no_rows} * (j-1) + \text{data_size} * (i-1)$$

where `ar2` = address of `ar2(1,1)`

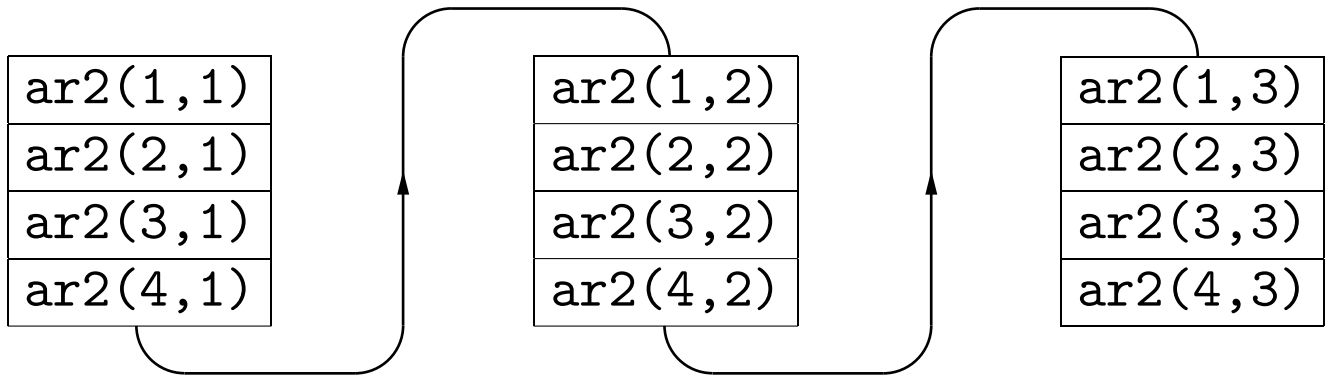
ARRAY STORAGE ORDER IN C (2-D)

```
float ar2[4][3];
```



ARRAY STORAGE ORDER IN FORTRAN (2-D)

DIMENSION AR2(4,3)



```

# init2d.s
# SPIM code to initialize the elements of a 2-d array
# with the value stored in location val

.data
ar2:      .word 0:12      # array of 12 integers (4 bytes each)
val:      .float 1.e-3    # value to store in array elements
nrows:    .word 4        # no. of rows
ncols:    .word 3        # no. of columns
size:     .word 4        # size of an array element, in bytes

# Register usage
# For clarity, each register holds only one variable
# Nobody would program in this way, because some registers can be re-used
#
# $t0      base address (address of ar2[0][0])
# $t1      size = size of an array element (in bytes)
# $t2      nrows = number of rows
# $t3      ncols = number of columns
# $t4      nrmax = nrows - 1 = max. value of row index
# $t5      ncmax = ncols - 1 = max. value of column index
# $t6      rindex = row index
# $t7      cindex = column index
# $t8      addr = absolute address of ar2[rindex][cindex]
# $t9      roffset = rindex * ncols * size
# $s0      coffset = cindex * size
# $s1      offset = roffset + coffset
# $s2      nc = column counter to be decremented
# $s3      nr = row counter to be decremented
# $s4      value to be stored in each array element

.text
__start:
la $t0, ar2
lw $t1, size
lw $t2, nrows
lw $t3, ncols
addi $t4, $t2, -1 # nrmax
addi $t5, $t3, -1 # ncmax
ori $t6, $0, 0    # initialize row index to 0
lwcl $f0, val
mfcl $s4, $0

rloop: mul $t9, $t6, $t3 # multiply rindex by ncols
mul $t9, $t9, $t1 # multiply by size of one array element to get roffset
ori $t7, $0, 0    # initialize column index to 0
cloop: mul $s0, $t7, $t1 # multiply cindex by size to get coffset
add $s1, $s0, $t9 # offset of ar2[rindex][cindex] = roffset + coffset
add $t8, $s1, $s1, $t0 # address of ar2[rindex][cindex] = offset + base
sw $s4, 0($t8)    # store val in ar2[rindex][cindex]
addi $t7, $t7, 1  # increment the column index
sub $s2, $t5, $t7 # nc = ncmax - cindex
bgez $s2, cloop  # branch back to cloop if nc >= 0
addi $t6, $t6, 1  # increment the row index
sub $s3, $t4, $t6 # nr = nrmax - rindex
bgez $s3, rloop  # branch back to rloop if nr >= 0
ori $v0, $0, 10  # reach here if row loop is done
syscall
# end of program!

```

ASSEMBLED TEXT FOR init2d.s

```

[0x00400000] 0x3c081001 lui $8, 4097 # la $t0, ar2
[0x00400004] 0x3c011001 lui $1, 4097
[0x00400008] 0x8c29003c lw $9, 60($1) # lw $t1, size
[0x0040000c] 0x3c011001 lui $1, 4097
[0x00400010] 0x8c2a0034 lw $10, 52($1) # lw $t2, nrows
[0x00400014] 0x3c011001 lui $1, 4097
[0x00400018] 0x8c2b0038 lw $11, 56($1) # lw $t3, ncols
[0x0040001c] 0x214cffff addi $12, $10, -1 # addi $t4, $t2, -1
[0x00400020] 0x216dffff addi $13, $11, -1 # addi $t5, $t3, -1
[0x00400024] 0x340e0000 ori $14, $0, 0 # ori $t6, $0, 0
[0x00400028] 0x3c011001 lui $1, 4097
[0x0040002c] 0xc4200030 lwcl $f0, 48($1) # lwcl $f0, val
[0x00400030] 0x44140000 mfc1 $20, $0 # mfc1 $s4, $0
[0x00400034] 0x01cb0018 mult $14, $11
[0x00400038] 0x0000c812 mflo $25 # mul $t9, $t6, $t3
[0x0040003c] 0x03290018 mult $25, $9
[0x00400040] 0x0000c812 mflo $25 # mul $t9, $t9, $t1
[0x00400044] 0x340f0000 ori $15, $0, 0 # ori $t7, $0, 0
[0x00400048] 0x01e90018 mult $15, $9
[0x0040004c] 0x00008012 mflo $16 # mul $s0, $t7, $t1
[0x00400050] 0x02198820 add $17, $16, $25 # add $s1, $s0, $t9
[0x00400054] 0x0228c020 add $24, $17, $8 # add $t8, $s1, $t0
[0x00400058] 0xaf140000 sw $20, 0($24) # sw $s4, 0($t8)
[0x0040005c] 0x21ef0001 addi $15, $15, 1 # addi $t7, $t7, 1
[0x00400060] 0x01af9022 sub $18, $13, $15 # sub $s2, $t5, $t7
[0x00400064] 0x0641ffff bgez $18 -28 # bgez $s2, cloop
[0x00400068] 0x21ce0001 addi $14, $14, 1 # addi $t6, $t6, 1
[0x0040006c] 0x018e9822 sub $19, $12, $14 # sub $s3, $t4, $t6
[0x00400070] 0x0661ffff bgez $19 -60 # bgez $s3, rloop
[0x00400074] 0x3402000a ori $2, $0, 10 # ori $v0, $0, 10
[0x00400078] 0x0000000c syscall # syscall

```

DATA STRUCTURES (7)

- A **pointer array** is an array whose elements are pointers
 - ▷ The memory locations that are pointed to don't have to be contiguous
 - ▷ A pointer array `par2[]` can be used to access the rows of a 2-dimensional array `ar2[][]`
 - `address of ar2[i][j] = par2[i] + size * j`
where `par2[i] = address of ar2[i][0] = addr. of row i`
 - A pointer-based array program uses more memory than a program that does index computation
 - Pointers result in a lower instruction count
 - ◊ To access `ar2[i][0] . . . ar2[i][n_cols-1]`, increment the pointer by `data_size` at each step (avoids multiplication!)

```

# init2dp.s
# SPIM code to initialize the elements of a 2-d array
# with the value stored in location val, using pointers
.data
ar2:      .word 0:12      # array of 12 integers (4 bytes each)
val:      .float 1.e-3    # value to store in array elements
nrows:    .word 4        # no. of rows
ncols:    .word 3        # no. of columns
size:     .word 4        # size of an array element, in bytes

# Register usage
# For clarity, each register holds only one variable
# Nobody would program in this way, because some registers can be re-used
#
# $t0      base address (address of ar2[0][0])
# $t1      size = size of an array element (in bytes)
# $t2      nrows = number of rows
# $t3      ncols = number of columns
# $t4      nrmx = nrows - 1 = max. value of row index
# $t5      ncmx = ncols - 1 = max. value of column index
# $t6      prow = pointer to 1st element in row
# $t7      pelem = pointer to current array element
#          = absolute address of ar2[rindex][cindex]
# $t8      bytes = no. of bytes in 1 row = ncols * size
# $s2      nc = column counter to be decremented
# $s3      nr = row counter to be decremented
# $s4      value to be stored in each array element

.text
__start:
    la $t0, ar2      # get pointer to start of array
    or $t6, $t0, $0  # initialize pointer to 1st row
    lw $t1, size
    lw $t2, nrows
    lw $t3, ncols
    mul $t8, $t3, $t1 # no. of bytes in 1 row = ncols * size
    addi $t4, $t2, -1 # nrmx
    addi $t5, $t3, -1 # ncmx
    or $s3, $t4, $0   # initialize row counter to nrmx
    lwcl $f0, val
    mfcl $s4, $0
    rloop: or $t7, $t6, $0 # initialize pointer to 1st element of 1st row
           or $s2, $t5, $0 # initialize nc to ncmx
    cloop: sw $s4, 0($t7)  # store val in ar2[rindex][cindex]
           add $t7, $t7, $t1 # increment the column pointer by the size of 1 element
           addi $s2, $s2, -1 # decrement nc by 1
           bgez $s2, cloop # branch back to cloop if nc >= 0
           add $t6, $t6, $t8 # increment the row pointer
           addi $s3, $s3, -1 # decrement nr by 1
           bgez $s3, rloop # branch back to rloop if nr >= 0
           ori $v0, $0, 10 # reach here if row loop is done
           syscall
           # end of program!

```

ASSEMBLED TEXT FOR init2dp.s

```

[0x00400000] 0x3c081001 lui $8, 4097 # la $t0, ar2
[0x00400004] 0x01007025 or $14, $8, $0 # or $t6, $t0, $0
[0x00400008] 0x01c07825 or $15, $14, $0 # or $t7, $t6, $0
[0x0040000c] 0x3c011001 lui $1, 4097
[0x00400010] 0x8c29003c lw $9, 60($1) # lw $t1, size
[0x00400014] 0x3c011001 lui $1, 4097
[0x00400018] 0x8c2a0034 lw $10, 52($1) # lw $t2, nrows
[0x0040001c] 0x3c011001 lui $1, 4097
[0x00400020] 0x8c2b0038 lw $11, 56($1) # lw $t3, ncols
[0x00400024] 0x01690018 mult $11, $9
[0x00400028] 0x0000c012 mflo $24 # mul $t8, $t3, $t1
[0x0040002c] 0x214cffff addi $12, $10, -1 # addi $t4, $t2, -1
[0x00400030] 0x216dffff addi $13, $11, -1 # addi $t5, $t3, -1
[0x00400034] 0x01809825 or $19, $12, $0 # or $s3, $t4, $0
[0x00400038] 0x3c011001 lui $1, 4097
[0x0040003c] 0xc4200030 lwc1 $f0, 48($1) # lwc1 $f0, val
[0x00400040] 0x44140000 mfc1 $20, $0 # mfc1 $s4, $0
[0x00400044] 0x01c07825 or $15, $14, $0 # or $t7, $t6, $0
[0x00400048] 0x01a09025 or $18, $13, $0 # or $s2, $t5, $0
[0x0040004c] 0xadf40000 sw $20, 0($15) # sw $s4, 0($t7)
[0x00400050] 0x01e97820 add $15, $15, $9 # add $t7, $t7, $t1
[0x00400054] 0x2252ffff addi $18, $18, -1 # addi $s2, $s2, -1
[0x00400058] 0x0641ffffd bgez $18 -12 # bgez $s2, cloop
[0x0040005c] 0x01d87020 add $14, $14, $24 # add $t6, $t6, $t8
[0x00400060] 0x2273ffff addi $19, $19, -1 # addi $s3, $s3, -1
[0x00400064] 0x0661ffff8 bgez $19 -32 # bgez $s3, rloop
[0x00400068] 0x3402000a ori $2, $0, 10 # ori $v0, $0, 10
[0x0040006c] 0x0000000c syscall # syscall

```

DATA STRUCTURES (8)

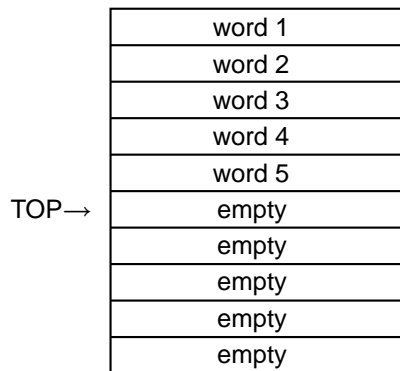
- A **list** is a data structure such that:
 - ▷ All elements are of the same data type
 - ▷ In a list with p elements, each element is labeled with one and only one of the integers $0, \dots, p - 1$ (one can also use $1, \dots, p$)
 - ▷ Example: A list of students enrolled in a computer architecture course
- A list can be implemented as an array
 - ▷ The elements of a list can be lists or other data structures
 - ▷ Common operations on lists:
 - Insert
 - Delete
 - Find k^{th} element

DATA STRUCTURES (9)

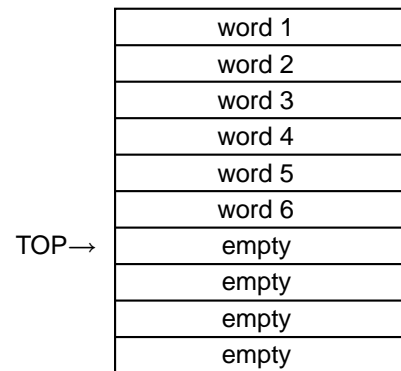
- A **linked list** is a list in which each element contains both data and the address of the next element
 - ▷ Can be implemented as a 2-dimensional array `ll [] []`
 - ▷ Each linked-list element is a row of the array
 - `ll [i] [0]` contains a pointer to the next element of the linked list
 - The data in the linked-list element is stored in
$$\text{ll [i] [1], \dots, ll [i] [n_cols-1]}$$
(can be elements of another data structure)
 - ▷ Common operations on linked lists:
 - Insert
 - Delete
 - Find k^{th} element

DATA STRUCTURES (10)

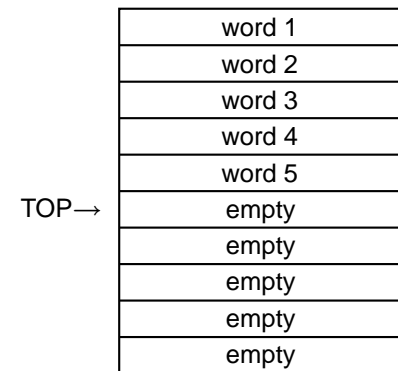
- A **stack** is a list in which insertions and deletions can be performed only in one position
 - ▷ The insertion/deletion position is called the **top** of the stack, even if the stack “grows” towards lower addresses
 - ▷ An insertion is called a **push**; a deletion is called a **pop**



STACK BEFORE WORD 6
IS PUSHED ON



STACK AFTER WORD 6
IS PUSHED ON



STACK AFTER WORD 6
IS POPPED OFF

TO PUSH: 1) COPY DATA TO TOP OF STACK
 2) INCREMENT STACK POINTER

TO POP: 3) DECREMENT STACK POINTER