Data Structures and MIPS

C data structures and their MIPS representations:
- char ... as byte in memory, or low-order byte in register
- int ... as one word in memory, or whole register
- double ... as two words in memory, or $f? register
- arrays ... sequence of memory bytes/words, accessed by index
- structs ... chunk of memory, accessed by fields (in C) or offsets (in MIPS)
- linked structures ... struct containing address of another struct

A char, int or double
- could be implemented in register if used in small scope
- could be implemented on stack if local to function
- could be implemented in .data if need longer persistence/lifetime

Static Allocation

Static allocation:
- uninitialised memory allocated at compile/assemble-time, e.g.
  ```
  int val;
  val: .space 4
  char str[20];
  str: .space 20
  int vec[20];
  vec: .space 80
  ```
- initialised memory allocated at compile/assemble-time, e.g.
  ```
  int val = 5;
  val: .word 5
  int arr[4] = {9,8,7,6};
  arr: .word 9, 8, 7, 6
  char *msg = "Hello\n";
  msg: .asciiz "Hello\n"
  ```

1-d Arrays in MIPS

Can be named/initialised as noted above:
```
vec: .space 40
# could be either int vec[10] or char vec[40]
nums: .word 1, 3, 5, 7, 9
# int nums[6] = {1,3,5,7,9}
```

Can access elements via index or cursor (pointer)
- either approach needs to account for size of elements

Arrays passed to functions via pointer to first element
- must also pass array size, since not available elsewhere

See sumOf() exercise for an example of passing an array to a function

1-d Arrays in MIPS

Scanning across an array of \(N\) elements using index
```c
# int vec[10] = {...};
# int i;
# for (i = 0; i < 10; i++)
# printf("%d\n", vec[i]);}

li $s0, 0  # i = 0
li $s1, 10  # no of elements
li $s2, 4   # sizeof each element
loop:
  bge $s0, $s1, end_loop  # if (i >= 10) break
  mul $t0, $s0, $s2  # index -> byte offset
  lw $a0, vec($t0)   # a0 = vec[i]
  jal print  # print a0
  addi $s0, $s0, 1  # i++
  j loop
end_loop:
```

Assumes the existence of a print() function to do printf("%d n",x)
1-d Arrays in MIPS

Scanning across an array of $N$ elements using cursor

```c
# int vec[10] = {...};
# int *cur, *end = &vec[10];
# for (cur = vec; cur < end; cur++)
#   printf("%d\n", *cur);
```

```assembly
la $s0, vec
la $s1, vec
loop:
   bge $s0, $s1, end_loop
   lw $a0, ($s0)
   jal print
   addi $s0, $s0, 4
   j loop
end_loop:
```

Assumes the existence of a print() function to do printf("%d n",x)

2-d Arrays in MIPS

2-d arrays could be represented two ways:

(a) ```c
int matrix[4][4];
```

(b) ![Diagram of 2-d Array Representation (b)]

Accessing elements:

Find **start** of row 1, then add **offset 2** within row

(a) ```c
x = matrix[1][2];
```

(b) ![Diagram of Accessing Elements (b)]

### 2-d Arrays in MIPS

Representations of `int matrix[4][4]` ...

(a) ```c
# for strategy (a)
matrix: .space 64
```

(b) ```c
# for strategy (b)
row0: .space 16
row1: .space 16
row2: .space 16
row3: .space 16
matrix: .word row0, row1, row2, row3
```

Now consider summing all elements

```c
int i, j, sum = 0;
for (i = 0; i < 4; i++)
   for (j = 0; j < 4; j++)
      sum += matrix[i][j];
```
2-d Arrays in MIPS

Computing sum of all elements in int matrix[6][5] in C

```c
int row, col, sum = 0;

// row-by-row
for (row = 0; row < 6; row++) {
    // col-by-col within row
    for (col = 0; col < 5; row++) {
        sum += matrix[row][col];
    }
}
```

2-d Arrays in MIPS

Computing sum of all elements for strategy (a) int matrix[6][5]

```assembly
li $s0, 0  # sum = 0
li $s1, 6  # s1 = #rows
li $s2, 0  # row = 0
li $s3, 5  # s3 = #cols
li $s4, 0  # col = 0  // redundant
li $s5, 4  # intsize = sizeof(int)
mul $s6, $s3, $s5  # rowsize = #cols*intsize

loop1:
    bge $s2, $s1, end1  # if (row >= 6) break
    li $s4, 0  # col = 0

loop2:
    bge $s4, $s3, end2  # if (col >= 5) break
    mul $t0, $s2, $s6  # t0 = row*rowsize
    mul $t1, $s4, $s5  # t1 = col*intsize
    add $t0, $t0, $t1  # offset = t0+t1
    lw $t0, matrix($t0)  # rowp = &matrix[i][0]
    loop1:
        beq $s2, $s1, end1  # if (row >= 4) break
        li $s3, 0  # col = 0
        mul $t0, $s2, $s1  # off = 4*i
        lw $s4, matrix($t0)  # rowp = &matrix[i][0]

loop2:
        beq $s3, $s1, end2  # if (col >= 4) break
        mul $t0, $s3, $s1  # off = 4*col
        add $t0, $t0, $s4  # int *p = &group[col]
        lw $t0, ($t0)  # t0 = *p
        add $t0, $s0, $t0  # sum += t0
        add $s3, $s3, 1  # col++
        j loop2

end2:
        addi $s2, $s2, 1  # row++
        j loop1

end1:
```

2-d Arrays in MIPS

Computing sum of all elements for strategy (b) int matrix[4][4]

```assembly
li $s0, 0  # sum = 0
li $s1, 4  # s1 = 4 (sizeof(int))
li $s2, 0  # row = 0

loop1:
    beq $s2, $s1, end1  # if (row >= 4) break
    li $s3, 0  # col = 0

loop2:
    beq $s3, $s1, end2  # if (col >= 4) break
    mul $t0, $s2, $s6  # t0 = row*rowsize
    mul $t1, $s4, $s5  # t1 = col*intsize
    add $t0, $t0, $t1  # offset = t0+t1
    lw $t0, matrix($t0)  # rowp = &matrix[i][0]
    loop1:
        beq $s2, $s1, end1  # if (row >= 4) break
        li $s3, 0  # col = 0
        mul $t0, $s2, $s1  # off = 4*i
        lw $s4, matrix($t0)  # rowp = &matrix[i][0]

loop2:
        beq $s3, $s1, end2  # if (col >= 4) break
        mul $t0, $s3, $s1  # off = 4*col
        add $t0, $t0, $s4  # int *p = &group[col]
        lw $t0, ($t0)  # t0 = *p
        add $t0, $s0, $t0  # sum += t0
        add $s3, $s3, 1  # col++
        j loop2

end2:
        addi $s2, $s2, 1  # row++
        j loop1

end1:
```