Data Structures and MIPS

C data structures and their MIPS representations:

- `char` ... as byte in memory, or register
- `int` ... as 4 bytes in memory, or register
- `double` ... as 8 bytes in memory, or `$f?` register
- arrays ... sequence of bytes in memory, elements accessed by index (calculated on MIPS)
- structs ... sequence of bytes in memory, accessed by fields (constant offsets on MIPS)

A `char`, `int` or `double`

- can be stored in register if local variable and no pointer to it
- otherwise stored on stack if local variable
- stored in data segment if global variable

Global/Static Variables

- global/static variables need appropriate number of bytes allocated in data segment using `.space`:

```
double  val;              val: .space 8
char str[20];             str: .space 20
int  vec[20];             vec: .space 80
```

initialized to 0 by default, other directives allow initialization to other values:

```
int val = 5;                 val: ..double 5
int arr[4] = {9,8,7,6};      arr: .word 9, 8, 7, 6
char msg[7] = "Hello\n";     msg: .asciiz "Hello\n"
```

add: local variables in registers

C

```
int main(void) {
    int x, y, z;
    x = 17;
    y = 25;
    z = x + y;
```

MIPS

```
main:
    # x in $t0
    # y in $t1
    # z in $t2
    li   $t0, 17
    li   $t1, 25
    add  $t2, $t1, $t0

    // ...
```

add: variables in memory

C

```
int x, y, z;
int main(void) {
    x = 17;
    y = 25;
    z = x + y;
```

MIPS

```
main:
    li   $t0, 17
    sw   $t0, x
    li   $t0, 25
    sw   $t0, y
    lw   $t0, x
    lw   $t1, y
    add  $t2, $t1, $t0
    sw   $t2, z
.data
x:  .space 4
y:  .space 4
z:  .space 4
```

store value in array element 1

C

```c
int x[10];

int main(void) {
    // sizeof x[0] == 4
    x[3] = 17;
}
```

MIPS

```
main:
    li   $t0, 3
# each array element
# is 4 bytes
    mul  $t1, $t0, 4
    la   $t1, x
    add  $t2, $t1, $t0
    li   $t3, 17
    sw   $t3, ($t2)
.data
x:  .space 40
```

store value in array element 2

C

```c
#include <stdint.h>

int16_t x[30];

int main(void) {
    // sizeof x[0] == 2
    x[13] = 23;
}
```

MIPS

```
main:
    li   $t0, 13
# each array element
# is 2 bytes
    mul  $t1, $t0, 2
    la   $t1, x
    add  $t2, $t1, $t0
    li   $t3, 23
    sw   $t3, ($t2)
.data
x:  .space 60
```

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

Printing 1-d Arrays in MIPS 1

C

```c
int vec[5]={0,1,2,3,4};
 // ...
 int i = 0
 while (i < 5) {
  printf("%d", vec[i]);
  i++;
 }
 // ....
```

- i in $s0

MIPS

```
    li   $s0, 0
loop:
    bge  $s0, 5, end
    la   $t0, vec
    mul  $t1, $s0, 4
    add  $t2, $t1, $t0
    lw   $a0, ($t2)
    li   $v0, 1
    syscall
    addi $s0, $s0, 1
    b    loop
end:
```

```
    .data
vec: .word 0,1,2,3,4
```

Printing 1-d Array in MIPS 2

C

```c
int vec[5]={0,1,2,3,4};
 // ...
 int *p = &vec[0];
 int *end = &vec[4];
 while (p <= end) {
  int y = *p;
  printf("%d", y);
  p++;
 }
 // ....
```

- p in $s0
- end in $s1

MIPS

```
    li   $s0, vec
    la   $t0, vec
    add  $s1, $t0, 16
loop:
    bgt  $s0, $s1, end
    lw   $a0, ($s0)
    li   $v0, 1
    syscall
    addi $s0, $s0, 4
    b    loop
end:
```

```
    .data
vec: .word 0,1,2,3,4
```

1-d Arrays in MIPS

Scanning across an array of `N` elements using cursor

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

   la   $s0, vec            # cur = &vec[0]
   la   $s1, vec+40         # end = &vec[10]
loop:
   bge  $s0, $s1, end_loop  # if (cur >= end) break
   lw   $a0, ($s0)          # a0 = *cur
   jal  print               # print a0
   addi $s0, $s0, 4         # cur++
   j    loop
end_loop:
```

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

2-d Arrays in MIPS

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

```
matrix: .space 64
```

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

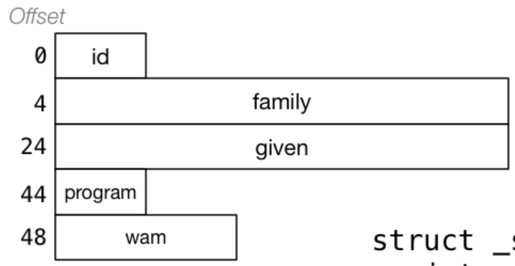
```c
int row, col, sum = 0;

// 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 `int matrix[6][5]`

```
   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) # t0 = *(matrix+offset)
   add  $s0, $s0, $t0    # sum += t0
   addi $s4, $s4, 1      # col++
   b    loop2
end2:
   addi $s2, $s2, 1      # row++
   b    loop1
end1:
```

Structs in MIPS




```
struct _student {
    int    id;
    char   family[20];
    char   given[20];
    int    program;
    double wam;
};
```

Structs in MIPS

C `struct` definitions effectively define a new type.

```
// new type called "struct student"
struct student {...};
// new type called student_t
typedef struct student student_t;
```

Instances of structures can be created by allocating space:

```
                 # sizeof(Student) == 56
stu1:            # student_t stu1;
     .space 56
stu2:            # student_t stu2;
     .space 56
stu:
     .space 4    # student_t *stu;
```

Structs in MIPS

Accessing structure components is by offset, not name

```
stu1: .space 56       # student_t stu1;
stu2: .space 56       # student_t stu2;
# stu is $s1          # student_t *stu;

li  $t0  5012345
sw  $t0, stu1+0       # stu1.id = 5012345;
li  $t0, 3778
sw  $t0, stu1+44      # stu1.program = 3778;

la  $s1, stu2         # stu = &stu2;
li  $t0, 3707
sw  $t0, 44($s1)      # stu->program = 3707;
li  $t0, 5034567
sw  $t0, 0($s1)       # stu->id = 5034567;
```