add: 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 $t0, y
    add $t2, $t1, $t0
    sw $t2, z
    // ...
.data
x: .space 4
y: .space 4
z: .space 4
```

store value in array element

C
```
#include <stdio.h>

int x[10];

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

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

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;
  char str[20];
  int vec[20];
  ```

  ```
  val: .space 4
  str: .space 20
  vec: .space 80
  ```

- initialised memory allocated at compile/assemble-time, e.g.
  
  ```
  int val = 5;
  int arr[4] = {9,8,7,6};
  char *msg = "Hello\n";
  ```

  ```
  val: .word 5
  arr: .word 9, 8, 7, 6
  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]
```

```
ums: .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

```
# int vec[10] = {...};
# int i;
# for (i = 0; i < 10; i++)
# printf("%d\n", vec[i]);}
```

```
li $s0, 0  # i = 0
li $s2, 4  # sizeof each element
```

```
loop:
  bge $s0, 10, 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++
```

```
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

```
# 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++
```

```
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

```
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

```
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

```
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, $s4, $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

j loop2

addi $s4, $s4, 1  # col++
addi $s2, $s2, 1  # row++
j loop1
end1:
```

Structs in MIPS

```
struct _student {
    int id;
    char family[20];
    char given[20];
    int program;
    double wam;
};
```
C struct definitions effectively define a new type.

```c
// 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:

```c
stu1: .space 56 # student_t stu1;
stu2: .space 56 # student_t stu2;
stu: .space 4 # student_t *stu;
```

Accessing structure components is by offset, not name:

```c
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;
lw $t1, 0($s1) # stu->program = 3707;
```

C can pass whole structures to functions, e.g.

```c
# Student stu; ...
# // set values in stu struct
# showStudent(stu);
... compute ...
# to access struct
addi $sp, $sp, 56 # pop st off stack
lw $ra, ($sp) # access struct
addi $sp, $sp, 4
lw $fp, ($sp)
addi $sp, $sp, 4
jr $ra # }
```
Structs in MIPS

Accessing struct within function ...

```asm
# start of function
showStudent:
# prologue

# function body
...
# t0 = s.id
lw $t0, 4($fp)
# t1 = s.prog
lw $t1, 48($fp)
# print s.family
la $a0, 8($fp)
l1 $v0, 4
syscall
...
```

Clearly a more efficient way to pass a large struct
Also, required if the function needs to update the original struct