Other type issues (arrays, structs...)

- we should examine the code needed to access array elements
- first, consider one-dimensional arrays, random access
- suppose our array indices start at base address B (usually 0 or 1, but could be any integer)
- given an array of N elements, each with size S
- the offset to element i is then (i-B)*S
- note that if B is 0 then this becomes simply i*S

1-D array, random access

- possible pseudo-asm to access Arr[i], where B=1, S=8

```
- loadAddr @Arr,R1 // copy Arr's addr to R1
- load i, R2        // copy i from mem to R2
- subi 1,R2         // subtract B from i
- multi 8,R2    // multiply by 8 to get offset to i
- add R1,R2,R2      // compute full address of A[i]
- load R2,R3        // finally, R3 := Arr[i]
```

Some optimizations

- possible pseudo-asm to access Arr[i], where B=1, S=8
  - `loadAddr @Arr,R1 // copy Arr's addr to R1`
  - `load i, R2       // copy i from mem to R2`
  - `// can get rid of subtract if B is 0`
  - `lshift 3,R2  // bit shift quicker than multiply`
  - `// often a combined load op availab, e.g.`
  - `loado R1,R2,R3    // R3 := Arr[i]`

1-dim, sequential access

- common to access a number of elements in order, e.g.
  - for i = a to b: Arr[i] = 0;
- instead of computing from scratch each time:
  - compute addr for Arr[a] normally
  - for each subsequent i, simply add the element size (e.g. 8)
- assumes elements are stored sequentially in memory

Multi-dim, sequential access

- `for r = m to n`
  - `for c = a to b`
    - `Arr[r][c] = 0`
- before first loop:
  - compute rowstart = addr of Arr[r][a]
  - offset=0
- at end of each pass through inner loop
  - add element size to offset
- at end of each pass through outer loop
  - add Arr's total row size to rowstart
  - set offset back to 0

Indirection vectors: 2-d

- each row of the array stored as one-dim array
- “outer” dimension of array is actually an array of pointers to the rows
  - Arr[i] points to i'th row
  - Arr[i][j] accesses j'th element of i'th row
- uses extra space (for the outer 1-d array), but rows no longer need to be contiguous

Access via indirection vector

- Arr actually holds address of indirection vector
- Suppose we want to iterate through all elements
  - `loadAddr @Arr, R0 // get addr of indir vector`
  - `repeat for each row`
    - `loadAddr R0,R1// get addr of row`
    - `loadi 0,R2 // offset to first element in row`
    - `repeat for each element in row:`
      - `loado R1,R2,R3 // R3 = Arr[0][0]`
      - `addi 8,R2`
    - `addi 8,R0 // R1 := addr holding start of next row`

Arrays as params

- usually passed as an address, even if goal is “by value”
- for multidim arrays, callee needs to know row sizes
- what if they're not known at compile time? e.g. dynamically allocated
- compiler can generate a descriptor: record of number of array dimensions and sizes
- compiler can pass a pointer to the descriptor as the array “parameter” (this ptr often called a dope vector)
- means adding extra code to callee to access descriptor

Character arrays as strings

- null terminated (O(n) to find end) vs store size (extra storage needed)
- access generally the same as for arrays
- what if a “char” is smaller than the smallest addressable size in asm
  - need to add bitmask operations around char access!
- concat(a,b):
  - either a:=a.b (simply copy b to end of a's space)
  - or return new string containing a.b
  - length(a.b):
    - concatenate first then compute
    - or compute length(a)+length(b)

Structures/records

- as discussed earlier, need to map order/offsets of fields, padding or re-ordering for alignment rules
- x.s might be implemented like
  - `loadAddr @X, R1`
  - `loadi 12, R2 // supposing 12 is the offset to S`
  - `loado R1,R2,R3 // R3 := x.s`

Arrays of structs

- could be implemented like programmer may expect, i.e. As an actual array of structs
- compiler could actually choose to implement it as a struct of arrays!
  - could provide more efficient sequential access when iterating across a fixed field
  - difference not visible to the programmer

Unions

- Allow programmer to specify a set of possible data types for the item
- Item given just enough memory to store largest of the data types in the set
  - `union num { int i; float f; } // num is int or float`
  - `num X;`
  - `x.i = 3; // store int 3 in x's memory space`
  - `x.f = 1.23; // overwrite x's space with float 1.23`
  - `int v = x.i; // use x's mem space as if type int,`
  - `              // copying contents to v`

Pointers and ambiguity (again)

- cold memory address/reference
- complicate compiler's ability to keep things in registers
  - *x could refer to same stored value as z, or as *y
  - could even refer to same stored value as *(y + 600)
- if compiler has each of the referenced values in registers then it can't know which ones are referring to the same "real" value
- has to take a safe/conservative approach and put things back in memory before/after uses