Announcements

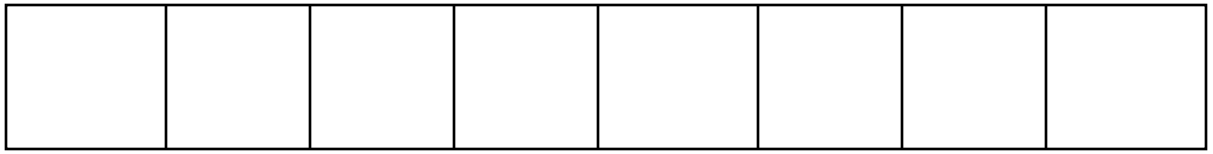
- Program #5
  - Due week from Today at 8:00 PM
- Reading
  - Bryant & O'Hallaron 3.8 (Today)
  - Bryant & O'Hallaron 3.7 (Tuesday)

Using gcov

- Must compile and link all program parts with
  - -fprofile-arcs -ftest-coverage
- Run program(s) to be tested for coverage
- Run gcov
  - gcov <file>.c
  - For the project this should be ast.c
- Look at the output file
  - <file>.c.gcov

Representing Arrays

- Arrays (C style)
  - sequence of consecutive memory locations
  - address of array is the starting point for the array
  - subscript defines offset into array
    - location = baseAddr + subscript * sizeof(object)
  - no information about array stored at runtime



baseAddr

Arrays in Other Languages

- Typical Implementations
- Java
  - sequence of objects
  - array descriptor (runtime data) stores
    - type information about what this is an array of
    - length array
- Fortran90
  - sequence of logically consecutive bytes
  - array descriptor (runtime data) stores additional information
    - distance between array elements (for each dimension)
    - number of elements (for each dimension)
    - lower bound of array dimension (for each dimension)

Comparison of Representations

- Runtime Descriptors
  - require space
  - allow checking of array overflow
  - permit passing multi-dimensional arrays as parameters
- Base Address Only
  - Simple
  - No extra space

Recall Computer from Project 3

- Recall general form of store and load:
  - lw Rn Rm <value>
    - Rn = memory[Rm + #<value>]
  - sw Rn Rm <value>
    - Memory[Rm + #<value>] = Rn
- Remember
  - R0 is always 0
  - bal Rn Rm label
    - saves currentPC + 1 into Rn

Example Array Access

```
for (i=0; i < 10; i++) {
    sum += a[i];
}
```

- For the machine from P1:

```
main:   li R3 1
        li R7 10
        neg R7
        li R4 0
        li R5 0
loop:   beq R7 R0 endLoop
        lw R6 R4 a
        add R6 R5 R5
        add R7 R3 R7
        add R4 R3 R4
        beq R0 R0 loop
endLoop: sw R5 R0 sum
        write R5
        halt
```

```
a:      .data 1
        .data 2
        .data 3
        .data 4
        .data 5
        .data 6
        .data 7
        .data 8
        .data 9
        .data 10
sum:    .data 0
```

Examples of Using Array Descriptors

- Can Use Additional information to Check array bounds
- Traditional C:

```
int strcmp(char *a, char *b) {
    int i;
    for (i=0; a[i] && b[i]; i++) {
        if (a[i] < b[i])  return -1;
        else if (b[i] < a[i])   return 1;
     }
     if (a[i] && !b[i]) return 1;
    else if (b[i]) return -1;
    else return 0;
}
```

With Array Descriptors

```
int strcmp(aRef *a, aRef *b) {
    int i;
    for (i=0; i < a->len && i < b->len && a[i] && b[i]; i++) {
        if (a[i] < b[i])  return -1;
        else if (b[i] < a[i])   return 1;
     }
    if (i >= a->len || i >= b->len) {
        printf("Array subscript error\n");
        exit(-1);
    }
     if (a[i] && !b[i]) return 1;
    else if (b[i]) return -1;
    else return 0;
}
```

Structures

- Store values in a sequence of memory locations
- Compiler remembers where the fields are stored
  - generates code to compute the field
    - fieldAddr = baseAddress + fieldOffset
- Example:

```
struct location {
    int x, y;
} ;

struct location spot = { 25, 36 };
main() {
    printf("%d\n", spot.x);
    printf("%d\n", spot.y);
}
```

```
main:   Load R3 R0 spot
        Output R3
        Load R2 R0 #1
        Load R3 R2 spot
        Output R3
        Halt
spot:   Data 25
        Data 36
```

Objects

- Very Similar to structures
- Include additional information
  - type of the object
  - functions to invoke on the object
    - sometime might know this from type info

Pointers

- Are simply memory locations storing other locations
- Get translated into load/stores via these locations
- Consider Walking a linked list (starting at head)

```
struct data {
    int x
    struct data *next;
};

void list(struct data *head)  {
   while (head) {
       printf("%d\",  head->x);
       head = head->next;
}
```

```
list:     lw R2 R0 head
          li R4 1
loop:     beq R2 R0 endList
          lw R3 R2 0
          write R3
          add R2 R4 R2
          lw R2 R2 0
          beq R0 R0 loop
endList:  halt
head:     .data 10
          .data 5
          .data 0
```