Allen Telescope Array
(SETI Institute)

Implementing Arrays

Lecture 8
CS 212 – Fall 2007

Arrays in Bali

- An array type is represented by a type followed by brackets
- Examples
  - int[ ] myIntegers,
  - char[ ] myCharacters,
- After these declarations, both myIntegers and myCharacters have the value *null*
- To initialize an array, assign an *array value*
- Array values
  - type[size]
    - Create array of given size
    - All elements have default value
  - type{exp1, exp2, exp3}
    - Each *expi* is an expression
    - Creates an array of size = number of expressions
- Examples: both produce arrays of size 4 holding zeros
  - myIntegers = int[4];
  - myIntegers = int{0, 0, 0, 0};

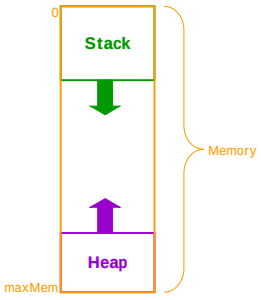
Multidimensional Arrays in Bali

- Multidimensional arrays can be created by adding more brackets
- Example declaration: int [ ] [ ] values;
- Example initializations
  - values = int [2] [3];
    - Produces 2-by-3 array of zeros
  - values = int { {1, 2, 3}, {4, 5, 6} };
    - Produces 2-by-3 array of integers
  - values = int{ {1}, {4, 5, 6} };
    - Produces two rows of varying length

Array Size in Bali

- To determine size (number of elements) of an array
  - Each array has a size "field"
- Examples
  - myIntegers.size
    - Produces the value 4
  - values.size
    - Produces the value 2

Stack vs. Heap



- Confusingly *Stack* and *Heap* are terms used both
  - for data structures and
  - for operating systems
- Typically have Stack start at one end of memory, Heap start at the other end
  - Stack and Heap collide implies Out-Of-Memory error

SaM's Heap

Implementing Bali Arrays

- Use the instruction MALLOC
  - Reserves space in the Heap
- Example sam-code

  PUSHIMM 4
  MALLOC
  - These instructions reserve a block of size 4 in SaM's Heap
    - 4 words for the array
    - There is some additional information stored in the Heap
      - You can mostly ignore this
      - SaM uses it to keep track of items in the Heap
  - MALLOC leaves the block's *address* on top of the Stack
    - This is the address of the word that holds the first array item
    - The array is located at *address*+0, *address*+1, *address*+2, and *address*+3

Code Patterns for Bali Arrays

Bali Code	Sam Code	Comment
myIntegers = int[4];	PUSHIMM 4 MALLOC STOREOFF 13	*Push array's size onto Stack* *Create heap-block for the array, and push block's address onto Stack* *We pretend myIntegers is at offset 13 from the FBR*
myIntegers[2] = 44;	PUSHOFF 13 PUSHIMM 2 ADD PUSHIMM 44 STOREIND	*Push array's address onto Stack* *Subscript* *Address arithmetic* *Stores 44 into myIntegers[2]*
x = myIntegers[2];	PUSHOFF 13 PUSHIMM 2 ADD PUSHIND STOREOFF 9	*Push array's address onto Stack* *Subscript* *Address arithmetic* *Stored value (44) placed on Stack* *We pretend x is at offset 9*

Use of null for Bali Arrays

- Declaring an array

  int[ ] A,
- Constructing an array

  A = int[6];
- Initializing an array

  i = 0;
  loop while i < 6;
  A[i] = i;
  i = i + 1;
  endloop
- When an array is declared but not yet constructed, the array variable has value null
- In the sam-code, an array variable (e.g., A) holds the address of the array
  - After array construction, this is an address in Heap
  - Before array construction, this should be an address clearly not within Heap (e.g., 0 works fine)
- In other words, null in Bali-code corresponds to 0 in sam-code

Arrays in Other Languages

- Bali arrays work much like Java arrays
  - Arrays are stored in the Heap
    - Array size is specified when array is *created* (via *new* in Java)
- Other choices
  - Arrays are allocated before the program runs (e.g., as in early Fortran)
    - Implies that each array is of *fixed size*
  - Arrays are stored on the Stack
    - Implies that array-size must be known when array is declared

Pointers

- Java hides pointers (but they're there)
- Pointers are used explicitly in C (and many other languages)
- A pointer is basically an *address* (of a cell in memory)
  - In Java, these addresses refer only to cells in the Heap
  - In C, these addresses can refer to *any* cell
- Pointer operations
  - Dereferencing: identify the thing that is pointed to
  - Assignment: copy pointer values
  - Comparison: equality/inequality of pointers
  - Dynamic allocation: a "new" block of memory
  - Deallocation: return a block of memory to the system
  - Arithmetic: used in C (mostly for arrays)

Pointers in C

- The code

  ```
  int *p;
  ```

  declares a variable p that can point to an integer
  - Immediately after declaration, it doesn't point at anything in particular
- This code

  ```
  int i, j, *p;
  p = &i;
  ```

  causes p to point at i

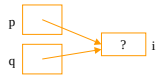
- * is the *indirection* operator
- & is the *address* operator
- These assignments are the same

  ```
  j = *&i;
  j = i;
  ```
- These are the same, too

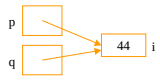
  ```
  i = 4;
  *p = 4;
  ```

C Pointer Examples

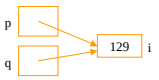
```
int i, j, *p, *q;
p = &i;
q = p;
```



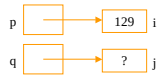

```
*p = 44;
```




```
*q = 129;
```




```
q = &j;
```

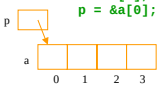


What about
`*q = *p;` vs. `q = p;` ?

Pointers and Arrays in C

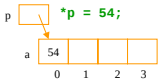
- A pointer can point at an array

```
int a[4], *p, *q;
p = &a[0];
```



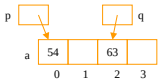
- You can use pointer arithmetic to access array elements

```
*p = 54;
```



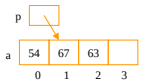
- Addition works

```
q = p + 2;
*q = 63;
```



- So does subtraction

```
p = &a[3];
p = p – 2;
*p = 67;
```



Oddities of Pointers and Arrays in C

- An array name can be used as a pointer

```
int a[4];
*a = 7;
*(a+1) = 77;
```

- A common way to sum the elements of an array

```
for (p = a; p < a+N; p++)
      sum += *p;
```

- These two references are the same:

```
a[i]
*(a + i)
```

- Also, strangely, these two are the same

```
a[i]
i[a]
```

because both are equivalent to `*(a + i)`

- Arrays and pointer are nearly equivalent, but you can't assign to an array name

Pointers in Java

- Java doesn't use pointers in an explicit way
  - Java implicitly uses pointers (called *references* in Java)
  - Every variable that does not hold a primitive type holds a reference (a pointer) to an Object
- There is no Java equivalent to the pointer arithmetic typically done in C
- In Java

```
Thing x;
```

declares that x holds a reference to an Object of type Thing

- The code

```
x = new Thing(...);
```

reserves space for an Object of type Thing in the Heap, initializes the Object, and places a reference to the object in x

Allocating/Deallocating Heap Memory

- In C
  - Allocating memory
    - malloc: allocates a block of memory (no initialization)
    - calloc: allocates a block of memory and clears it
    - realloc: resizes a previously allocated block of memory
  - Deallocating memory
    - free(p): deallocates block of memory that p points to
    - Beware of *dangling pointers*!
- In Java
  - Allocating memory
    - The *new* operator
      - allocates a block of memory
      - calls the specified *constructor*
  - Deallocating memory
    - Java uses an automatic *garbage collector*
      - frees any allocated memory that is no longer in use
    - Can choose to run it using the System.gc method

Runtime Data Areas

- For SaM
  - Code
  - Stack
  - Heap
  - Registers
- For Java
  - Method area
  - Java stacks
  - Heap
  - PC registers
  - Native method stacks

from: http://www.artima.com/insidejvm/ed2/jvm2.html

JVM Runtime Data Areas

- Method area (stores data for each type)
  - Information about the type (e.g., name, modifiers, superclass, etc.)
  - *Constant pool* for the type
    - Any constant used in the type's code (e.g., 5 or 'x' or 1.414)
  - Field & method information for the type (including the *code* for each method)
  - *Class variables* (i.e., *static* fields)
- Java stacks
  - Stores *stack frames*
  - But keeps *multiple* stacks because Java is *multithreaded*
- Heap
  - Stores objects (including *instance variables*)
- PC registers
  - One PC register for each *thread*
- Native method stacks
  - A work area for methods written in a language other than Java