Week 10
Pointers and the Heap

Paul Chew
CS 212 – Spring 2004

Announcements

- Part 3 updates
  - Due date has been delayed by one week
    - Now due: Monday, April 19, 11pm
  - Be sure you handle all parts of the Part 3 grammar!
- Sections are meeting
  - Today
  - Next week, too
- Make use of Office Hours!

- If your Part 2 did not compile or if it failed many tests
  - The graders are not expected to determine the exact nature of any problems with your code
  - If there is some small error, you can request a regrade
    - Describe the problem
    - Provide working code
  - Do not use this as a coding strategy — penalties increase for Parts 3 and 4

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
  ```c
  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
    ```c
    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
    ```c
    j = &i;
    j = i;
    ```
  - These are the same, too
    ```c
    i = &j;
    *p = 4;
    ```

C Pointer Examples

- A pointer can point at an array
  ```c
  int a[4], *p, *q;
  p = a[0];
  ```
  - Addition works
    ```c
    q = &a[2];
    *q = 63;
    ```
  - You can use pointer arithmetic to access array elements
    ```c
    *p = 54;
    ```
  - So does subtraction
    ```c
    p = a[3];
    p = p - 2;
    ```

Pointers and Arrays in C

- If your Part 2 did not compile or if it failed many tests
  - The graders are not expected to determine the exact nature of any problems with your code
  - If there is some small error, you can request a regrade
    - Describe the problem
    - Provide working code
  - Do not use this as a coding strategy — penalties increase for Parts 3 and 4
Oddsities of Pointers and Arrays in C

- An array name can be used as a pointer:
  ```c
  int a[4];
  *(a+1) = 7;
  ```
- A common way to sum the elements of an array:
  ```c
  for (p = a; p < a+N; p++)
      sum += *p;
  ```
- These two references are the same:
  ```c
  a[i]  
  *(a + i)
  ```
- Also, strangely, these two are the same:
  ```c
  a[i]  
  i[a]
  ```
- 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
  Thing x;
  ```
- Java implicitly uses pointers (called references in Java):
  ```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:
  ```java
  Thing x;
  ```
  ```java
  x = new Thing(...);
  ```
- In Java:
  ```java
  Thing x;
  ```
  ```java
  x = new Thing(...);
  ```

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(): 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
    - 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

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