Pointers

Week 9
CS 212 – Spring 2008

Reminder

- Project Part 3
  - Design Document is due on Thursday, March 27
  - Part 3 code is due Thursday, April 10

What to Put in Your Design Document

- Specify each class
  - For each class, specify the class's methods
    - For each method, specify
      - Its arguments (i.e., its interface)
      - Its preconditions (if any)
      - Its postconditions (i.e., what the method does)
- Specify how the classes interact
  - Diagrams can be useful here, but aren't required
    - UML (Unified Modeling Language) can be used, but informal diagrams are OK, too
- Expected length of design document
  - One page ⇒ probably too short
  - Ten pages ⇒ definitely too long

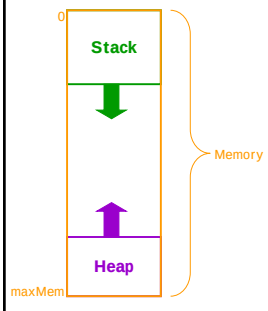
Motivating Dynamic Allocation

- Some programming matches well with a Stack
  - Temporary values used to evaluate an expression
  - Local variables
  - Stack frames for function calls
- But not everything we want to do has stack-like behavior
  - Maintain a linked list during insertions and deletions
  - A function to read a file and return the resulting graph (vertices & edges)
  - Build a binary search tree
- We need a place to store an arbitrary number of items, each of arbitrary size

Dynamic Allocation

- Heap
  - Place for dynamic allocation
  - Allows you to store large "things" without having to push/pop the Stack
- Creating items in the Heap
  - Claim a chunk of Heap-memory
  - Store address of that chunk
  - Use that address to access your item
  - When done with item, eliminate it from Heap, making chunk of Heap-memory available for re-use

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

Using SaM's Heap

- SaM memory allocation: MALLOC
  - Pops top of Stack
  - Allocates that number of memory cells in heap
  - Pushes the address of the first heap-cell onto stack
- SaM example:

```
PUSHIMM 1       // 1 cell to allocate
MALLOC          // pop 1 and allocate 1 cell in heap
PUSHIMM 3       // 3 cells to allocate
MALLOC          // pop 3 and allocate 3 cells in heap
PUSHIMM 0       // no cells to allocate
MALLOC          // pop 0 and allocate no cells in heap
FREE            // deallocate last "object"
FREE            // deallocate second "object"
FREE            // deallocate first "object"
PUSHIMM 0       // push dummy return value
STOP            // cease execution
```

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

Garbage Collection

- Want to keep any object that can be reached from program's variables
  - Either directly or through other objects that can be reached
  - *Program's variables* = anything in the *call stack*
- Once "not-in-use" objects are found
  - Can reclaim the memory for re-use
  - Can also compact memory
    - I.e., move all the "in-use" objects to another memory block (without gaps between objects)

Garbage Collector Schemes

- Mark and Sweep
  - Mark every object as "not-in-use"
  - Starting from the call stack, visit every reachable object, marking it as "in-use"
  - Everything still marked "not-in-use" can be reclaimed
- Reference Counting
  - Every object keeps a count of how many pointers reference it
  - When count is zero, memory can be reclaimed
  - Problem: cycles!
- For either scheme
  - Can "stop the world"
  - Can interleave (i.e., take turns)
  - Can run concurrently
- Java's current garbage collector
  - A 2-tier scheme (old generation; new generation)
  - A mark-and-sweep method
  - With compaction
- Java's garbage collection scheme has changed as new Java versions were released

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)

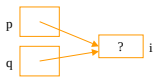
Pointers in C

- 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;
  ```

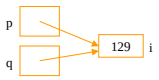
C Pointer Examples

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

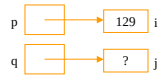



```
*p = 44;
```


```
*q = 129;
```




```
q = &j;
```

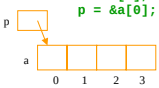


What about
`*q = *p;` vs. `q = p;` ?

Pointers and Arrays in C

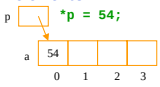
- A pointer can point at an array

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



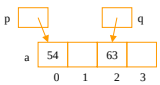
- You can use pointer arithmetic to access array elements

```
*p = 54;
```



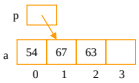
- Addition works

```
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

Where do Arrays Live?

- 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

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

GBA Runtime Data Areas

Memory Map (Simplified)

BIOS
Work RAM
Control Registers
Palettes
Video Display
Sprites
Game Code

- BIOS (Basic Input/Output System)
  - System stuff; normally inaccessible
- Work RAM
  - Workspace; variables are stored here
- Control Registers
  - Setting these alters the way the game is displayed
- Palettes
  - Used to compactly represent colors
- Video display
  - Data here is displayed on the game-screen
- Sprites
  - These are small images that can be layered on top of the video display
- Game code