Announcements:
- P2 and grades <= 40
- guest speaker on 11/19 (Prof. Paul Francis!)
- Hitsheh and section
- P3 due date...
- “no compile” → bad!

Reading:

Overview:
- motivation
- SaM and The Heap
- the heap is not necessarily a heap, though it is a “heap”
- “basics” of C and pointers
- pointers and memory allocation and the heap

---

1. Introduction
1.1 “Under the Covers”
- what’s Java doing?
- what actually happens to objects?
- how to write a grammar for classes?
- how to write a compiler for objects?

1.2 Roadmap for Part 4
- computer architecture revisited: main memory
- simulating objects: pointers
- memory map: heap
- C and pointers
- JVM, SaM: memory map
- objects in sam-code with MALLOC
- Bali++, compiler

1.3 Questions
- How does an object get into the heap?
- How does Java find the object?
- How does an object use correct method and data?
- How does an object get garbage collected?

---

2. Computer Memory
2.1 Main Memory
- store information as bits (bytes, nibbles, words, …)
- may think of MM as array

2.2 Cells
- memory split into cells
- cells have address
- cells store values

2.3 Values
- can be actual values (integers, bytes, …)
- addresses of other cells (pointers)
- pointer:
  - address of a “thing” in memory
  - usually address of object in memory

---

3. Simulating Objects
3.1 Related example
- doubly-linked list:

```
list → head → a → b → c → d → e → f → g
              DLL  DLN  DLL  DLN  DLL
```
- areas of memory “point” to other areas of memory
- what if didn’t have objects with references?

3.2 Use Arrays!
- extremely handy because:
  - elements have addresses
  - resembles cells in main memory
- cells of array:
  - contain values
  - contain “pointers”
- pointers are addresses of other cells
3.3 Array Implementation
- example) create DLL (see 3.2)
- try a single array
- need to partition array into list “nodes”
- must store data in node and links to prev and next
- need notion of null: use out of bounds value

   list → a ← b ← c ← =

   data next prev nil (-1)

   list → 0 1 2 3 4 5 6 7 8 9 10

- multiple arrays are helpful

4. Heap
4.1 Modern Languages
- objects go into “the heap”
- how to store objects?
- use pointer concept!
- what’s “the heap”?

4.2 General Heap
- nothing to do with programming
- pile of many things

4.3 Data Structures Heap
- See http://www.nist.gov/dads/HTML/heap.html
- “A tree where every node has a key more extreme (greater or less) than or equal to the key of its parent.”

4.4 The Heap
- refer to memory map for modern languages:

   static area  program area

   stack        heap

- the heap: memory reserved for dynamic allocation
  - cells store values
  - cells have addresses
- just like array model from before!

5. Pointers
- want to demonstrate how cells refer to each other
- provides access to addressable memory
- Java: internally pointers happening…

5.1 Pointer Type
- variable has value of memory address or nil
- nil: cannot access any memory cell
- SaM: we use 0 address (where no object lives)

5.2 Uses of pointer
- efficiency: move pointer to large data structure, not the structure itself
- dynamic data: help to create, build, destroy structures

5.3 Pointer Operations
- dynamic allocation: create data structure
- dereferencing: identify the “thing” pointed to
- assignment: copy pointer values
- comparison: equality/inequality of pointers
- deallocation: remove data structure
- arithmetic? (C, C++)
6. C Pointers

6.1 Basics

- **pointer**: address of “thing” in memory
- usually declare a variable as a pointer to another type
- need a special operation to find address of a pointer

6.2 Declaration

- `type *var`
- `var` holds a pointer (memory address) to a variable of type `type`
- examples)
  - `int x; /* type: int */`
  - `int *y; /* type: pointer to int */`

6.3 Finding Address

- `&`: address-of operator
- `&var`
  - returns address of `var`
  - read as “the address of `var`”
  - typically store value of `&var` in a pointer
- example)
  ```c
  int x, *y; /* int, pointer to int */
  x = 1; /* x = 1 */
  y = &x; /* y gets address of x */
  ```

6.4 Dereferencing

- `*`: 
  - dereferencing operator (“point-to operator”)
  - also called indirection operator
- `*var`: has the value of what `var` points to
- if `var` points to another variable, *that* variable’s value is returned

6.5 Arithmetic

- can perform arithmetic on pointers
- makes a pointer point elsewhere...

6.6 Example 1

```c
/* pointers1.c */
int main(void) {
  int x, *y;
  x = 1;
  y = &x;
  printf("%s%i
","  x: ", x);
  printf("%s%i
"," &x: ", &x);
  printf("%s%i
","  y: ", y);
  printf("%s%i
"," *y: ", *y);
  return 0;
}
/* Output:
x: 1
&x: -4262456
y: -4262456
*y: -4262460
*/
```

<table>
<thead>
<tr>
<th>address</th>
<th>var</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4262456</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>-4262460</td>
<td>y</td>
<td>-4262456</td>
</tr>
</tbody>
</table>

6.7 Example 2

```c
int main(void) {
  int i;
  int *j;
  printf("%s
","i, *j declared:);
  printf("%s%i
","  i: ", i);
  printf("%s%i
"," &i: ", &i);
  printf("%s%i
","  j: ", j);
  printf("%s%i
"," &j: ", &j);
  printf("%s%i
"," *j: ", *j);
  return 0;
}
/* Output:
i, *j declared:
i: 0
&i: -4262632
j: -4262240
&j: -4262636
*j: 0
*/
```
6.8 Example 3

```c
int main(void) {
    int a, b, c;
    int *pa;
    int *pb;
    pa = &a;
    pb = &b;
    b = 1;
    pa = pb;
    c = *pa;
    printf("c: %i\n", c);
    return 0;
}
```

7. Memory Allocation

7.1 Arrays

- treated as pointers in C/C++
- need to allocate memory from heap
- example:
  ```c
  int b[10];
  int *a; int n = 4;
  a = malloc(n, sizeof(int));
  ```
- some weirdness:
  - `char c[10]` means that `c` is a constant pointer that points to `c[0]`
  - can’t change `c` (e.g., arithmetic)
- more weirdness:
  - `p[i]` and `*(p+i)` are identical
  - `*(p+i)`:
    deference a pointer that points `i` positions past `p`
- why the weirdness? allocation of memory from heap!

7.2 Structures/Records

- composite data types
- think classes without methods
  ```c
  struct Complex {
    float R;
    float I;
  } Complex;
  ```
- can `struct`s of `struct`s, too!
- member access:
  - `struct_var.member_name`
  - `pointer_to_struct -> member_name`

7.3 Arrays of Structures

- need to allocate enough space for each element
  ```c
  Complex * complexVec(int n) {
    Complex * v;
    v = (Complex *)malloc((n)*sizeof(Complex));
    return v;
  }
  ```
- note use of `malloc` for later!

7.4 Example

```c
#include <stdlib.h>
typedef struct {
    float R;
    float I;
} Complex;

Complex complex(float real, float imag);
Complex * complexVec(int n);
void print(Complex * c, int length);

int main(int argc, char* argv[]) {
    Complex * c;
    c = complexVec(2);
    c[0] = complex(0,1);
    c[1] = complex(1,2);
    print(c,2);
    free(c);
    return 0;
}

Complex complex(float real, float imag) {
    Complex c;
    c.R = real;
    c.I = imag;
    return c;
}
```
Complex * complexVec(int n) {
    Complex * v;
    v = (Complex *)malloc((n)*sizeof(Complex));
    return v;
}

void print(Complex * c, int length) {
    int i;
    for (i=0;i<length;i++)
        printf("%f%s%f%s\n", c[i].R, " ", c[i].I, "i");
}

/* Output:
0.000000+1.000000i
1.000000+2.000000i */

8. More Pointers

8.1 Pointers in C and C++
- hold the actual addresses of data in memory
- can be cast to different data types
- can be altered to point to other memory locations
- allows programmer to corrupt all kinds of things by accessing and manipulating memory cells!
- Java Pointers? no! safer, but more restrictive...

8.2 References
- C++
  - reference type: special kind of pointer type
  - constant pointer that is always implicitly dereferenced
  - must be initialized with address of a variable
- Java Reference
  - Similar to C++ reference
  - holds the memory address of an object
  - reference values can not be altered
  - references can only be recast to a super or sub-class of that object, never to other data types

9. The JVM Architecture

9.1 Three Versions of the Java Virtual Machine
- abstract specification: http://java.sun.com/docs/books/vmspec/
- concrete implementation: what you use to program
- runtime instance: created to run a specific program

9.2 JVM Architecture
- Chapter 3 of JVM specification

9.3 Class Files
- everything you know about Java syntax
- essentially, specification on what can compile and run

9.4 Class Loaders
- find and load types
- neat stuff to see in java.lang.ClassLoader

9.5 Execution Engine
- resembles our SaM
- the “brain” behind the JVM
- native calls? System.out.print ...
9.6 Area Descriptions

- **pc register**:
  - program counter
  - contain address of non-native JBC instruction for a particular thread

- **JVM stacks**:
  - hold local vars, partial results
  - help with method invocation and returning
  - essentially, “the stack”

- **heap**:
  - store all class instances and arrays (objects)
  - shared by all threads

- **method area**:
  - holds field & method data, method & constructor code
  - shared by all threads
  - “program area” from earlier/CS211 notes

- **runtime constant pool**:
  - constants, literals

- **native method stacks**:
  - stack to support methods written in different languages, such as C

10. Java Performance

10.1 Sturgeon’s Law

- “Ninety percent of everything is crap.”
- http://www.jargon.net/jargonfile/s/SturgeonsLaw.html
- Computers and 80/20 (or 90/10)
  - processor spends large amount of time processing a small amount of code
  - improving such hot spots helps!

10.2 Java Options

- http://java.sun.com/j2se/1.4.2/docs/tooldocs/tools.html
- command-line java options:
  - Unix: `man java` and `java -help`
  - Windows: http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/java.html

10.3 Java Profiler

- identify what’s allocating heap
- try `java -Xrunhprof:help`

10.4 java.lang.Runtime

- Java application has a single instance of class `Runtime`
- application interfaces with environment in which the application runs
- use `getRuntime`

10.5 Memory

- `java -msx` sets the startup size of the memory allocation pool (the garbage collected heap) to x.
- `java -mx` sets the maximum size of the memory allocation pool (the garbage collected heap) to x.

// Inspired by http://tutorials.findtutorials.com/
// read/query/Java/id/216/p/1
import java.util.Vector;
public class HeapKiller {
    public static void main(String[] args) {
        System.out.println("Max: "+Runtime.getRuntime().maxMemory());
        System.out.println("Total: "+Runtime.getRuntime().totalMemory());
        System.out.println("Before: "+Runtime.getRuntime().freeMemory());
        Vector v = new Vector();
        while(true) {
            v.add("new int[1000]");
            System.out.print("Size: "+v.size());
            System.out.print("Total: "+Runtime.getRuntime().totalMemory());
            System.out.println("Free: "+Runtime.getRuntime().freeMemory());
        }
    }
}

10.6 Improving Performance

- http://tutorials.findtutorials.com/read/query/Java/id/214
- use native methods
- used `Buffered<>...` I/O classes
- avoid creating new objects
- do `String` concatenations with `StringBuffer`
- `ArrayList` instead of `Vector`
- `HashMap` instead of `Hashtable`
- make methods `final`, `static`, or `private` (forces static binding at compile time)
- see more at the website…