CSE431 – Translation of Computer Languages

- Runtime Support

Doug Shook

Runtime Operations

- How it used to be:
  - No memory organization
    - Memory was just a big array of bytes
    - Up to the programmer to access appropriate locations
    - Code == data

Static Allocation

- Every object in memory was set to a fixed address
  - Sometimes addresses would be *overlayed*
  - Pros?
  - Cons?
- Still used to an extent today
  - For what?

Stack Allocation

- Consider recursion:
  - Each recursive call requires a new allocation of memory (why?)

- Because of the above, we cannot determine how much memory is required at compile time
  - How do we manage these dynamic allocations?

Stack Frames

- When a procedure is compiled, the compiler records
  - Total memory required by the procedure
    - Machine specific (why?)
  - Offset of parameters / local values within procedure
    - Also machine specific

- Why is this information necessary?

- Literals are typically *not* stored in a stack frame
  - Where then?
  - Why?

Field Access

- Given a reference to a class instance, how can we look up field variables?

- What about methods?

Field Access

- Examine the following:

```
class k {
    int a;
    int sum(){
        int b = 42;
        return a+b;
}   }
```

- In the sum method:
  - Where does a come from (in memory)?
  - What about b?

Arrays

- Simplest array: static and one dimensional

```
int a[100]
```

- What factors affect the size of an array?

- Many processors impose alignment restrictions
  - Might require the use of padding

- How can we compute the address of an array element at position i?

Arrays – Dynamic Arrays

- Bounds not determined until runtime:

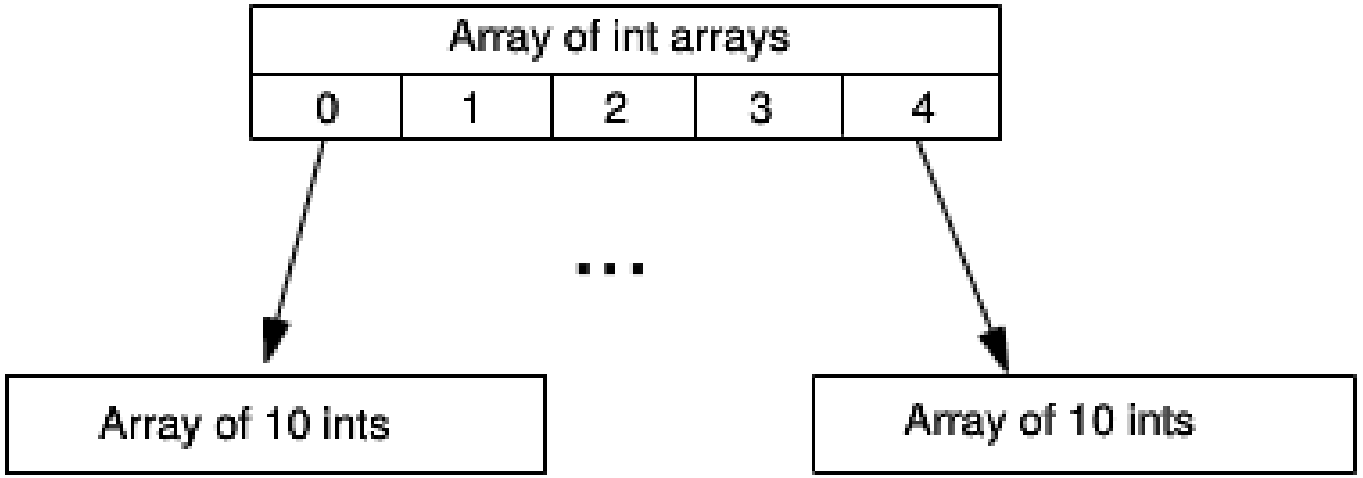
```
int a[numGrades()];
```

- Since we don't know the space at compile time, we can't allocate space for it
  - Where will it end up?

- Flex arrays take this a step further
  - Bounds can change during runtime

Arrays – Multidimensional

- Technically an array of arrays:



- Alternatively: one large block

A[0][0]	A[0][1]	...	A[1][0]	A[1][1]	...	A[9][8]	A[9][9]

Heap

- In Java
  - Local variables and methods are on the stack
    - Might include references to object instances
  - Instances and other objects are on the heap
    - Must have a way to dynamically allocate/deallocate heap space

Heap Allocation

- Explicit
- Implicit
  - Str1 + Str2

- Many heaps implement a free space list
  - What blocks are not in use?
  - Allocation searches the list for a contiguous block of sufficient size

- Problem: must combine contiguous blocks as they are deallocated
  - See text for two approaches

Heap Allocation

- When allocating, which block to grab?
  - Want to be efficient
    - Both processing and memory

- Allocation Schemes
  - Best fit
  - First fit
  - Next fit
  - Segregated Free Space Lists
  - Subheaps

Heap Deallocation

- Values on the heap that are no longer used are called dead objects
  - No longer accessible
  - What problems may arise?

- Deallocation Schemes
  - Manual
  - Garbage collection

Garbage Collection

- Attempts to track heap usage
  - Automatically deallocate when no longer in use
- Schemes
  - Reference counting
  - Mark-sweep
  - Copying Collectors
  - Generational
  - Conservative

Register Allocation

- Problem: we have a limited number of registers
  - Want to reduce memory accesses (why?)
- How many registers do we need for a particular piece of code?
  - Depends on live ranges of variables

Register Allocation

- Once we have the live ranges we can construct an interference graph
  - Variables that are alive at the same time are connected by an edge

```
int x = 7
int y = 2;
do.something(x);
int z = 9;
if(z)
   do.something.else();
this.thing = y;
```

Register Allocation

- Register allocation is now a graph coloring problem
  - No two vertices can have the same color
  - Up to k colors, where k = ?
- Problem: this is NP complete

Register Allocation

- Heuristic:
  - Find a vertex with edges < k
  - Push vertex on stack, remove from graph
  - Continue until no more vertices remain or no vertices with edges < k
    - Which do we prefer?
  - Start popping vertices and adding them back to the graph
  - Color each vertex as it is re-added

Exercise

- Try coloring the following graph for k = 3:

