Memory Management

Part 2

Data Types

• Values held in machine locations
• Integers, reals, characters, Booleans are built into languages as primitive types
  – Machine location directly contains the value
  – Efficiently implemented, likely understood by the instruction set
• Others built on top of them: structured types
  – Laid out in sequence of locations in the machine
  – Arrays, records, pointers.
  – Hopefully can be treated as first class citizens
    • A first class citizen can be passed as a parameter, returned from a subroutine, or assigned into a variable.
Arrays

- A sequence of elements of the same type stored consecutively in memory
- Element can be accessed quickly [O(1)]
- Accessed via indexing
  - A[i] : i→ index
- Index is often an integer but does not have to be
  - Must be efficiently computed
  - Here we are not including “associative” arrays that are really more like hash tables
- When is array bound computed?
- When is the space for the array allocated?
- Where is the space for the array allocated?
  - Java: from the Heap

Array Initialization

- Should the values in an array be pre-initialized?
  - Java initializes all values to 0 or null
  - C/C++ do no initialization, array contains whatever values happen to be sitting in memory
- Issue of efficiency
Arrays in Pascal

- May have any range of indices
  array [21-30] of real
- May have non integer indexes
  array [(Mon, Tue, Wed, Thu, Fri)] of integer;
  array [char] of token;
  type token = (plus, minus, times, divide, number,
  lparen, rparen, semi);
- These non-integer values really map to integer
  values internally for efficiency purposes
  – E.g. Mon=0, Tue=1, Wed=2, etc.

Arrays

- Should array type include bounds?
- Pascal did and it causes some problems
  – typeof(A[10]) ≠ typeof(A[100])
- Function arguments with arrays are problematic
  – Sort function with an array size of size 10 can’t
take array of size 9
  – Instead must pass array bounds as parameters
Arrays
Layout

- Determines the machine address of the i’th element relative to the address of the first element
- Different from allocation
  - Reserve actual machine memory for the array
- The elements of the array appear in consecutive locations

Arrays
Layout (C-Like Language)

```
int[] A = new int[10];
γ(A[i]) = γ(A[0])+e*i
0 <= i < n
```
e=element size, i=index

Strongly typed language requires checking type in dope vector
Arrays

var A : array [low .. high] of T

• base
  – Starting address of the first element A[low]
• width
  – size of an element of type T
• The elements are stored at
  – base, base+width, base + 2*width ....
• Address of A[i] computed in 2 parts
  – Compile time : offset from base
  – Run time : location of base

Arrays

• Address of A[i]
  = base +(i-low)*width
  = i*width + (base-low*width)
• (base-low*width) may be precomputed and stored
  – This is generally the value associated with an array variable
• i*width : must be computed at runtime
• If low = 0
  – Address of A[i] = i*width + base
• Time to compute the address is independent of i
  – So we get O(1) or constant access time
Multidimensional Arrays

- Common in all languages
  - C: A[200][200]
- Allocated in linear fashion
- Row major
  - Store by rows: row 1, row 2, row 3, ....
- Column major
  - Store by columns

```
char[][] C = new char[4][3];
y(C[i][j]) = y(C[0][0]) + e*(ni+j)  
0 <= i < m and 0 <= j < n
```
Multidimensional Arrays

• Address of M[i][j]

\[ \text{base} + (i - \text{low}_1) \times w_1 + (j - \text{low}_2) \times w_2 \]

- \( w_1 \) : width of a row = \( w_2 \times n_2 \)
- \( w_2 \) : width of an element
- \( n_1 \) : number of elements in a column
- \( n_2 \) : number of elements in a row = \( \text{high}_2 - \text{low}_2 + 1 \)

• Fixed part : \( \text{base} - \text{low}_1 \times w_1 - \text{low}_2 \times w_2 \)

• Variable part : \( i \times w_1 + j \times w_2 \)

Multi-D Arrays (Java)

• Java actually stores only 1D arrays; multi-dimensional arrays are references to other arrays

int[][][] nums = new int[4][3];
Type Checking for Arrays

- Create new typemap

\[
\text{typing}(\text{Declaration } d) = \bigcup_{i \in \{1..n\}} \begin{cases} <d_i.v,d_i.t,d_i.size> & \text{if } d_i \text{ is an ArrayDecl} \\ <d_i.v,d_i.t> & \text{otherwise} \end{cases}
\]

Check type of the array

\[
V(\text{Declarations } d) = \forall i \in \{1..n\}:
\begin{cases}
  d_i \in \text{ArrayDecl} \to 0 < d_i.size \land d_i.t \in \{\text{int, float, boolean}\} \land \\
  \forall j \in \{1..n\} : i \neq j \to d_i.v \neq d_j.v
\end{cases}
\]

Type Checking for Arrays

- Must also check validity of Expressions and Statements

\[
V(\text{Expression } e, \text{Typemap } tm) =
\begin{cases} 
  e \in tm \land \text{typeof}(e.index) = \text{int} & \text{if } e \text{ is an ArrayVariable} \\
\end{cases}
\]

\[
V(\text{Statement } s, \text{Typemap } tm) =
\begin{cases} 
  \text{typeof}(s.arg.e, index, tm) = \text{int} & \text{if } s \text{ is an Assignment and } s\.target \text{ is an ArrayVariable} \\
\end{cases}
\]
Semantics for Arrays

- Skipping info in the text regarding details of allocate and deallocate

\[ M(\text{Expression } e, \text{State } \sigma) = \sigma(e) \quad \text{if } e \text{ is an ArrayVariable} \]
\[ \land \quad 0 \leq e.\text{index} < \sigma(e.\text{size}) \]

\[ \sigma(v) = \mu(\gamma(v) + v.\text{index}) \quad \text{if } v \text{ is an ArrayVariable} \]
\[ = \mu(\gamma(v)) \quad \text{otherwise} \]

\[ M(\text{Assignment } s, \text{State } \sigma) \]
\[ = \sigma \cup \{ < \mu(\gamma(s.\text{target})) + M(s.\text{target.\text{index}}, \sigma), M(s.\text{source}, \sigma) > \} \]
\[ \text{if } s.\text{target} \text{ is an ArrayVariable} \]

For arrays, variable name replaced by memory address

Structures / Records

- Multiple values grouped together
- Heterogeneous elements allowed, no longer homogeneous as an array
- Variables relevant to an object to be grouped together and treated as a unit

Examples

typedef struct {
    int id;
    char[25] name;
    int age;
    double salary;
    char dept;
} employeeType;

class employeeType {
    int id;
    String name;
    int age;
    double salary;
    char dept;
} employeeType;

employeeType employee;
If using Java strings, which are actually objects, then instead of “Robert Noonan” there would really be a reference to another place on the heap where “Robert Noonan” is stored.

- Memory allocation at compile time
- Operations to access them
  - Dot operator x.re, x.im;
  - Assignment?
    - PASCAL allows
    - Java?
      - What happens if we assign one class to another?
    - C / C++?
      - Copies data but does not follow any pointers
      - Same as Java if classes in C++
Dangling Pointers

• Structures are often used as nodes within dynamic data structures, such as linked lists
• Raises the possibility of the **dangling pointer**
  – A pointer to storage used for another purpose and the storage is subsequently deallocated
• Garbage
  – Allocated but inaccessible memory locations
• Programs that create garbage are said to have *memory leaks*

Dangling Pointer Example

class node {
    int value, node next
};
node p, q;
p = new node();
q = new node();
q = p;
delete(p);

(a) ![Diagram of node structure](image)
(b) ![Diagram of node structure](image)
(c) ![Diagram of node structure](image)

“dangling reference”
“orphan”
Memory Leak Terms

• Dangling reference/Widow
  – A pointer to storage used for another purpose and the storage is subsequently deallocated

• Garbage/Orphan
  – Allocated but inaccessible memory locations

• Programs that create garbage are said to have memory leaks

Avoiding Garbage

• Many languages ask the programmer to explicitly manage the heap, where memory is allocated
  – C, C++,…
  – User must make sure to destroy everything that is allocated
  – Memory management is generally not central to the problem the programmer is trying to solve
  – What if something is missed? Easy to do…

    void foo()
    {
        p = new node();
        if (b) return;
        delete(p);
    }

• Interpreted and functional languages generally do automatic garbage collection
  – Java, Lisp,…
Garbage Collection

- Motivation from functional programming
- Increased importance due to OOP

*How do we reduce/eliminate the burden of memory management from the programmer?*

Garbage Collection Algorithms

- Reference counting
- Mark-Sweep
- Copy collection
- In Java
  - The garbage collector runs as a low-priority thread. It is automatic but it can be explicitly called by: System.gc (regardless of the state of the heap at the time of the call).
Garbage Collection
Reference Counting

• Free List
  – Heap is a continuous chain of nodes called the free list
  – Each node has an extra field to keep a count as well as a field to keep track of the node size

• Reference Count
  – Number of pointers referencing that node
  – Initially set to 0

Node creation via new()
  – Get nodes from the free list
  – Set reference count to 1

• Pointer Assignment
  – e.g. p=q;
  – Increment the reference count of q by 1
  – Decrement the reference count of p by 1
    • If zero, nothing references p so it is safe to delete
      – must also decrement reference count for any pointer in p’s data area by one. If one of these counts becomes zero, repeat for it’s descendants
      – Destroy p
  – Then perform the assignment
Garbage Collection
Reference Counting

• Pointer Deletion
  – e.g. delete p;
  – Decrement p’s reference count
    If refcount == 0
      For every pointer q in p’s data area
        delete q
      Put p on the free list
    Set p to null

Garbage Collection
Reference Counting

• The algorithm is activated dynamically on
  – new
  – delete

• Advantages
  – Very simple, fast, non-compacting garbage collection
  – Heap maintenance spread throughout program execution
    (instead of suspending the program when the garbage collector runs)
  – Must not forget to adjust reference counts on any pointer assignment
    (including passing pointers as subroutine arguments), or disaster can happen
Reference Counting Example

\[
\text{node } p, q, t; \\
p = \text{new node();} \\
q = \text{new node();} \\
p.\text{next} = q; \\
t = \text{new node();}
\]

if \( p.\text{next} \) pointed to something, we'd decrement the ref

Reference Counting Example

\[
t = \text{new node();} \\
t.\text{next} = q; \\
delete q; \\
q = \text{new node();}
\]
Reference Counting Example

```cpp
q = new node();
p = q
q = t
```
Garbage Collection
Reference Counting

- Minor Problem – Storage overhead for reference count
- Major Problem - Can’t handle circular chains of nodes

```java
p.next=null;
```

Reference Counting
Garbage Collection
Mark-Sweep

• Unlike reference counting, called when the heap becomes full
  – i.e. free list becomes empty
• Orphans are reassigned to the free list
  – Possibly large number of nodes
  – May be time consuming
  – Advantage over reference counting is it reclaims all garbage, even those in circular chains
• 2 Pass algorithm
  – 1<sup>st</sup> pass: Mark all the nodes if they are accessible
  – 2<sup>nd</sup> pass: Reassign the orphans

Garbage Collection
Mark-Sweep

• Mark Phase
  – Start with the active variables
  – Follow the links and “mark” the nodes that can be accessed
  – All unmarked nodes are orphans
• Sweep Phase
  – Follow all nodes in the heap
  – If the node is unmarked return to free list
  – Unmark all nodes that were not returned
Garbage Collection
Mark-Sweep

Online Demo

- Heap of Fish
- http://www.artima.com/insidejvm/applets/HeapOfFish.html
**Garbage Collection**

**Mark-Sweep**

- **Advantages**
  - Not invoked unless needed
    - Small programs don’t need it
    - Typically perform a large number of new/delete before this is needed
  - Reclaims all garbage
    - No problem with circular chains
  - Reduced memory overhead
    - Integer vs. a bit

- **Disadvantages**
  - Time consuming when used
    - 2 pass algorithm

**Garbage Collection**

**Copy Collection aka Stop and Copy**

- Time-space compromise compared to Mark-Sweep
- Also invoked only when heap becomes full
- Significantly faster than Mark-Sweep
  - Only 1 pass over the heap
  - But heap size is effectively reduced by half
    - i.e. copy collection uses a lot more memory, (but this is not as bad as it sounds if using virtual memory, can still have data in all available physical memory)
Garbage Collection

Copy Collection

- Divide the heap into two equal halves
  - *from_space*: All active nodes are kept here.
  - *to_space*: Used as a copy buffer

  When the *from_space* becomes full
  - All accessible nodes are copied into *to_space*
    - The descendents are copied as well
    - Copying to the *to_space* called *Forwarding*
      - Everything in the *from_space* is then added to the free list
  - Swap the roles of *from_space* and *to_space*
  - Eliminates the inaccessible nodes
  - Skipping some details here of allocating nodes from the free list of the *to_space*
Efficiency of Copy Collection vs. Mark Sweep

- M = heap size
- R = amount of live memory
- r = R/M is the residency
- m = amount of memory reclaimed
- t = time needed for reclaiming memory
- e = m/t is the efficiency of garbage collection (memory reclaimed per time)

Efficiency Continued

- Comparison:
  \[ t_{\text{copy}} = aR \quad t_{\text{MS}} = bR + cM \]
  \[ m_{\text{copy}} = \frac{M}{2} - R \quad m_{\text{MS}} = M - R \]
  \[ e_{\text{copy}} = \frac{M}{2aR} - \frac{1}{a} = \frac{1}{2ar} - \frac{1}{a} \]
  Since \( r < 1 \), copy collection better for small \( r \)
  \[ e_{\text{MS}} = \frac{M - R}{bR + cM} = \frac{1 - r}{br + c} \]
  As \( r \) increases, mark sweep becomes more efficient (as \( r \) approaches \( M/2 \))
Garbage Collection Today

- Many newer, complex algorithms proposed
- Active area of research
  - Incremental garbage collectors
  - Efficient garbage collectors (e.g., no recursion)
  - Generational garbage collectors
    - Separate objects that are in a young/old generation; older are more likely to survive, so might only scan younger generations, condemn older generations less frequently
- Hard to judge algorithm in isolation
  - Often must consider hardware considerations such as paging, virtual memory