Memory Management Functions

Chapter 9

Memory Management

- Process of binding values to memory locations
- Values may be static or dynamic
- Values are assigned at different places
  - Static memory
  - Run-time stack
  - Heap
Memory Management

Static Memory

• Allocated to values whose storage requirements are known at compile time

• Remain constant throughout the life of the program

• May be global or local to functions

Memory Management

Run-time Stack

• Pivotal structure in activation of methods

• Activation
  – A stack frame is pushed on top of stack

• Deactivation
  – A stack frame is popped from the stack

• Storage space for
  – Local variables
  – Actual parameters
  – Return values
Memory Management

Heap

- Storage for all dynamically allocated memory
- More unstructured
- Allocation and deallocation may happen in arbitrary order
  - Memory may become fragmented
  - Need for garbage collection
  - We will describe garbage collection algorithms

Typical Run-Time Memory Structure
Run-Time Memory Structure

- Static memory remains fixed throughout
- Stack grows and shrinks in structured manner
  - Stack pointer keeps track of the beginning of current stack
  - Stack size usually defined at compile time
- Heap h usually defined at the beginning of run time
  - Some programs use little or no heap space
  - Some machines allow h to vary
  - Accessed via pointers or references

Run-Time Memory Structure

- Address space
  - Range of addresses available to the program to use
  - Likely logical as opposed to physical
- Range of addresses
  - 0…n
  - Static area starts with 0
  - Top of the stack: a
  - Beginning of the heap : h
Run-Time Memory Structure

• h and n are defined at the beginning of runtime
• Size of heap space may vary
  – Based on the need of the program
• Size of stack depends on the runtime behavior
  – Level of nesting/recursion
    \[ 0 \leq a \leq h < n \]
• Otherwise the dreaded “stack overflow” error

Run-Time Memory Structure

• Must manage memory explicitly when using stacks and heaps
• Defined using an
  – Environment for an active method
    • Pairs of variable names and memory addresses accessible within the method
  – Memory map
    • Pairs of memory addresses and stored values
    • Addresses may be unused (locations to which no variables have been allocated) or undefined (locations allocated a variable but no value yet assigned)
Run-Time Memory Structure

\[
\begin{align*}
\int i &= 13, j = -1, k; \quad \text{inside method } m \\
\gamma_m &= \{<i,154>,<j,155>,<k,156>\} \\
\mu &= \{<a-1,\text{undef}>,<a,\text{unused}>,...<n,\text{unused}>\} \\
\sigma_m &= \gamma_m \times \mu \\
\text{Address of a variable is given by } \gamma_m(v): \\
\text{Value of a variable is given by } \mu(\gamma_m(v)) = \sigma_m(v)
\end{align*}
\]

\[
\begin{align*}
\gamma_m(i) &= 154 & \gamma_m(j) &= 155 \\
\sigma_m(i) &= \mu(\gamma_m(i)) = 13 & \sigma_m(j) &= \mu(\gamma_m(j)) = -1
\end{align*}
\]

Run-Time Memory Structure

- Where can the memory location for a variable be?
  - Static area
  - Stack
  - Heap
    - (accessed via pointer)
- Addressing function for static and stack variables must return a value between 0 and \( a - 1 \)
  \[
  0 \leq \gamma_m(v) \leq a - 1
  \]
Run-Time Memory
Allocate (using Stack)

- Done using declarations
  - Assume each variable can fit in a single memory location

\[ allocate(d_1, d_2, \ldots, d_k, \sigma) = \gamma \times \mu' \]
\[ \gamma' = \gamma \cup \{ <v_1, a>, <v_2, a+1>, \ldots, <v_k, a+k-1> \} \]
\[ \mu' = \mu \bigtriangleup \{ <a, \text{undef}>, <a+1, \text{undef}>, \ldots, <a+k-1, \text{undef}> \} \]
\[ a' = a + k \]

- Environment update uses regular set union
  - Allows variables with same names but different memory locations
- Memory map update uses overriding set union
- Well-defined as long as \( a + k < h \)

Run-Time Memory
Deallocate (using Stack)

\[ deallocate(d_1, d_2, \ldots, d_k, \sigma) = \gamma \times \mu' \]
\[ \gamma' = \gamma - \{ <v_1, a>, <v_2, a+1>, \ldots, <v_k, a+k-1> \} \]
\[ \mu' = \mu \bigtriangleup \{ <a, \text{unused}>, <a+1, \text{unused}>, \ldots, <a+k-1, \text{unused}> \} \]
\[ a' = a - k \]

- Environment update uses regular set difference
- Memory map update uses overriding set union
  - Real systems often skip this step for purposes of efficiency, will just be overwritten with the next allocate
Methods, Procedures, Functions

- Lots of benefits for using methods
  - Abstraction
    - Define a procedure “ComputeIt” that hides internal details
  - Implementation Hiding
    - Modify innards of a method without having to change the calling code
  - Modularity
    - Smaller pieces better understood
    - Easier to develop in isolation
  - Libraries
    - Extending the language, e.g. mathematical functions or graphics

Methods Activation

- Invoking a method

- Method P calls Method Q
  - P is put on hold
  - Control is passed to Q
    - Q generally has access to its own limited scope of variables
  - Q quits
  - Control is sent back to P

- Recursive: multiple activations of same method
Memory Management

- All memory needed for activation must be allocated dynamically.
- Stack is typically used
- Limitation
  - Imperative languages require procedures to be declared up front with parameters.
  - So we can’t create functions on the fly, e.g. have the program create its own functions and invoke them

Typical Procedure Activation

- Caller evaluates the actuals and places the value in the activation record for callee
- The state information is saved (to return to caller)
- Callee allocates space
  - Local variables
  - Temporary storage (expressions : intermediates)
- Execute the body of callee
  - May call another procedure (next frame)
Procedure Activation

- Control returns to the caller
  - Return values is placed so caller can find it
    - On stack or in registers
  - Restore the old state
  - Return control to caller
  - Pop the stack (get ready for the next call)

Activation Record or Stack Frame

- Locals
- Arguments
- Static link
  - Link to the static area
- Dynamic link
  - Stack frame of caller
- Other data to store
  - Return Address
  - Function result
  - Temp Storage
## Layout of Activation Records (C)

Local variables are referenced as an offset from the Frame Pointer. Computed at compile time.

### Method Activation

**Example**

**Static Scoping:**
Scope determined by program’s static structure at compile time.
**Method Activation**

**Example**

Dynamic scoping easier to see here e.g. for variable i in Frame B
Considered dangerous today due to side effects

**Static Variables**

- Retain their values between activations
- Storage for them is allocated statically at compile time
- Non-static local variables have dynamic memory allocation at activation time
Example: Examine Stack for the C Program

```c
int bar(int x)
{
    int z=5;
    return z;
}

int foo(int x)
{
    int y=3;
    x = x + y;
    y = bar(x);
    return x;
}

int main(int argc, char* argv[])
{
    int a=1, b=2, c=3;
    b = foo(a);
    printf("%d %d %d\n",a,b,c);
    return 0;
}
```

Formal Description of Stack Allocation

• For C-like language:
  – Value associated with an address can be
    • Unused, Undefined, Int, Boolean, Address
    • Address = Value that references another memory location, denoted by @r, where r is in {0…n}
  – Initial state for previous C-Like program
    • σ₁ = Ø × µ(0) = {<0, unused>, <1, unused>, …}
  – After declaring static variables
    • σ₂ = γ₂ × µ =
      {<b,0>, <c,1>} × {<0, undef>, <1, undef>, <2, unused>, …}
Stack Allocation

• After main begins execution:
  - \( \sigma_{main} = \gamma_{main} \times \mu \)
  - Where \( \gamma_{main} = \{ <h,0>,<i,1>,<slink,2>,<dlink,3>,<a,4>,<b,5> \} \)
  - \( \mu = \{ <0,undef>, <1,undef>, <2,@0>, <3,@0>, <4,undef>, <5,undef>, <6,unused>, ... <n,unused> \} \)
  - \( \gamma_{main} = \gamma_{main} \cup \{ <slink,@0>, <dlink,@0>, <a,undef>, <b,undef> \} \)

• State after call to any method \( m \), with \( np \) parms and \( nd \) locals:
  - \( \sigma_m = \text{allocate}(\text{slink}, \text{dlink}, m.\text{params}, m.\text{locals}, \sigma) \)
  - \( \gamma_m \times \mu \)
  - \( \gamma_m = \gamma_m \cup \{ <slink,a>, <dlink,a+1>, <p_1,a+2>, ... <p_{np},a+np+1>, <d_i,a+np+2>, ... <d_{nd},a+np+nd+1> \} \)

Stack Allocation

• Active memory:
  - \( \sigma(\text{slink}_{main}) = @0 \)
  - \( \sigma(\text{dlink}_{main}) = @a-1 \)
  - \( \mu = \mu(a) \cup \{ <a, @0>, <a+1, @a-1>, <a+2, v_1>, ... <a+np+1, v_{np}>, <a+np+2, undef>, ... <a+np+nd+1, undef>, <a+np+nd+2, unused>, ... <n, unused> \} \)
Parameter Passing

- Mapping between formals and actuals
- Call by value: pass the value
  - Value parameters
  - e.g., passing primitives in Java
- Call by reference: pass the address
  - Reference parameters
  - e.g., passing objects in Java... actually also call by value, since the value of an object is really its address
- Call by name: pass the name as is
  - Name parameters

Call by Value

- Formal parameter corresponds to the value of the actual
- Evaluate the argument at the time of call
- Place the value in the stack corresponding to the argument
- No side effect w.r.t. arguments
- Easy to understand
- Primary passing mechanism in C, Pascal and most other languages
Call by Value Example

```c
swap(int x, y);
{
    int z;
    z = x;
    x = y;
    y = z;
}
```

- Does this work? Common mistake made in CS201
- No values are changed in calling procedure

Call by Reference

- Formal parameter becomes synonymous with location of the actual parameter
- This means the actual can’t be an expression
  - Must be a variable or an assignable component of a data structure
- Location is computed and is passed to the called procedure
Call by Ref Example

Say this is passed by reference

Invocation:
int a=1; b=2;
swap(&a,&b);

Call by Reference in C/C++

Invocation:
int a=1; b=2;
swap(a,b);

Invocation:
int a=1; b=2;
swap(a,&b);
Call by Reference

C

- **swap(&a,&b)**
  - Pass the address of a and b
- Only parameter passing mechanism in C is call by value
- Can get call by reference by passing pointers explicitly
  - The * operator is used to dereference a pointer

Call by Value-Result

- Copy in /copy out
- **Copy in phase**
  - Values and locations of actuals computed
  - Copy value to beginning of the activation record
  - At this point, just like call by value
- **Copy out phase**
  - Upon termination of the method, the final values from the storage location on the activation record are copied back to the original locations
  - End result is like call by reference
Semantics of Call and Return

- We can extend the meaning function $M$ for Clite to include Call and Return for methods
- $M(\text{Call } s, \text{ State } \sigma) = \text{deactivate}(s.\text{name}, M(s.\text{body}, \text{activate}(s.\text{name}, s.\text{args}, \sigma)))$
- In other words we perform the following steps
  - Activate the call of $s$ with arguments $s.\text{args}$, by creating a new stack frame and establishing the value/reference associations between arguments and parameters
  - Determine the meaning of the body $s$ created by the activation
  - Deactivate the call by removing the stack frame created in the first step
- See text for details… we already saw allocating a stack frame, to deallocate we remove the newly added elements to $\gamma$ and $\mu$ via set subtraction, set the old values in $\mu$ to unused

Pointers

- Commonly used in C,C++
- A pointer is a memory address, or reference, to some other variable
  - Defined using * by the variable
  - -> used to dereference (*) and access member of the structure
- Why?
  - Indirect access to data
  - Intended to point to objects of a specific type
  - Fixed size, independent of type; single machine location
  - Efficiency : Avoid move/copy large data structures
  - Dynamic Data : Allow data structures to grow and shrink during execution
C pointer example

- Definition of a Node

```c
struct Node {
    int key;
    Node *next;
};
```

Node *head, *temp;
head = (Node *)malloc(sizeof(Node));
temp = (Node *)malloc(sizeof(Node));
(*head).key = 1;  head->next = temp;
head->next->key = 3;
temp = (Node *)malloc(sizeof(Node));
head->next->next = temp;  head->next->next->key = 5;
head->next->next->next = NULL;
```

Pointers

... 
// Search for key x
Node *p = head;
while ((p != NULL) && (p->key != x))
    p = p->next;

Pointers often viewed as the bane of reliable software development
*p could point anywhere in memory, e.g.:
p = 0;
*p = 100;  // Change contents of mem[0] to 100
But allows for great efficiencies. How can we get by without pointers in Java?