CSCI 2320
Principles of Programming Languages

Functions and Memory Management
Reading: Ch 9 (9.5 - 9.7) & Ch 11
(Tucker & Noonan)

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.
-- B. Stroustrup

Memory architecture for PL
The Structure of Run-Time Memory

Major areas

- **Static area**
  - Fixed size, fixed content
  - Allocated at compile time

- **Run-time stack**
  - Variable size, variable content
  - Used for function call and return

- **Heap:**
  - Fixed size, variable content
  - **Dynamically** allocated objects and data structures
Stack and heap overflow

- *Stack overflow* occurs when the top of stack exceeds its fixed limit.

- *Heap overflow* occurs when a call to `new` occurs and the heap does not have a large enough block available to satisfy the call.

Run-time stack

Ch 9
Implementation of functions

- Issues
  - Parameter passing: by value vs. by reference
  - Only pass-by-value: C, Java, Python
  - Both pass-by-value and reference: C++
    - Example of pass-by-reference
      ```
      void change(int &x) { x = 10; }
      void main() { int a = 0; change(a); }
      ```
    - Misconception: mixing up pointers with pass-by-reference (see codes on Blackboard)
- How functions (are made to) work
  - Activation record/stack frame
  - Push-pop operations in run-time stack

Activation record/Stack frame

- A block of information associated with each function call
  - Static link - to the function’s static parent (only in a nested function)
  - Dynamic link - to the activation record of the caller
  - parameters and local variables
  - Return address - jump to the memory location of the next instruction (in caller) after this function finishes
  - Saved registers
  - Temporary variables - values of expressions like x + 1
  - Return value - like a local var, but copied to a well-known, shared space accessible to the caller
Activation record

Run time stack

- A stack of activation records
  - Each new call pushes an activation record, and each completing call pops the topmost one
  - So, the topmost record is the most recent call
  - The stack has all active calls at any run-time moment
Is one stack-frame per function enough?

- No. Recursive functions.
- A function that can call itself, either directly or indirectly, is a recursive function. E.g.,

```c
int factorial (int n) {
    if (n < 2)
        return 1;
    else return n*factorial(n-1);
}
```

Stack activity for factorial(3)

<table>
<thead>
<tr>
<th>Call</th>
<th>Stack</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>n=3</td>
<td></td>
</tr>
<tr>
<td>Second</td>
<td>n=3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>n=2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>n=1</td>
<td>1</td>
</tr>
<tr>
<td>Third</td>
<td>n=3</td>
<td>2*1=2</td>
</tr>
<tr>
<td></td>
<td>n=2</td>
<td>3*2=6</td>
</tr>
</tbody>
</table>

Simplified: return values not shown
Stack activity

Global variables in static area

```
int h, i;
void B(int w) {
    int j, k;
    i = 2*w;
    w = w+1;
}
void A(int x, int y) {
    bool i, j;
    B(h);
}
int main() {
    int a, b;
    h = 5; a = 3; b = 2;
    A(a, b);
}
```

Heap

Ch 11
Allocating heap blocks

The function `new` allocates a block of heap space to the program.

E.g., `new(5)` returns the address of the next block of 5 words available in the heap:

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>undef</td>
<td>12</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>undef</td>
<td>0</td>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
<td>undefined</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

How are arrays allocated?

Dynamic arrays (Java)

JVM spec does not mandate contiguous allocation, but usually arrays are allocated contiguously

```
int [] A = new int[10];
```

Does C allocate arrays in this fashion? (int x[10];)

Where do C’s arrays (local var) live?
Garbage collection
What is garbage?
How does it arise?
How to reclaim unused space?

Garbage

- Garbage is a block of heap memory that cannot be accessed by the program.

- Garbage can occur when either:
  - an orphan is created
  - a widow is created
Garbage example

class node {
    int value;
    node next;
}
node p, q;

p = new node();
q = new node();
q = p; //creates orphan
delete p; //creates widow

Garbage collection algorithms

Garbage collection is any strategy that reclaims unused heap blocks for later use by the program.

Origin: John McCarthy (1960) for LISP

Three classical garbage collection strategies:

- Reference Counting - occurs whenever a heap block is allocated, but doesn’t detect all garbage.
- Mark-Sweep - Occurs only on heap overflow, detects all garbage, but makes two passes on the heap.
- Copy Collection - Faster than mark-sweep, but reduces the size of the heap space.
Algorithm 1: reference counting

- **Activation**: new, delete, **assigning one pointer to another**
- **Data structure**
  - `free_list`: linked list of free blocks
  - Each block has an RC
- **Algorithm**
  - **Event 1**: creation of a new incoming edge
    - Increase the RC of the block by 1
  - **Event 2**: Deletion of an incoming edge
    - (a) Decrease the RC of the block by 1
    - (b) If the RC hits 0, add the block to the `free_list` and decrease the RC of its direct descendent by 1. Recursively apply (b) if the descendent RC becomes 0.

---

Example

```
Example

[p.next = null]
```

```
[p.next = null]
```

```
[p.next = null]
```
Corrections to textbook

- Reference counting is activated for new, delete, and any assignment of one pointer to another.
- Upon deletion of an incoming arrow, only the direct descendant’s RC is decremented by 1, not all descendents’ (or the chain of descendents’) RC.

Pros and cons

- Pros
  - Dynamic (triggered by certain operations)
- Cons
  - Cannot detect circularly referencing orphans
  - Storage overhead of storing the RCs
Algorithm 2: mark-sweep

- Activation: only when heap overflow occurs
- Data structure
  - Each node has a mark bit (MB), initially set to 0
- Two passes
  - Pass I: Mark all nodes that are (directly or indirectly) accessible from the stack by setting their MB=1.
  - Pass II: Sweep through the entire heap and return all unmarked (MB=0) nodes to the free list.

---

Pass I (Mark)

- Triggered by \( t = \text{new node}() \) and \( \text{free_list} = \text{null} \).
- All MBs are already initialized to 0
- Mark every block reachable from stack (e.g., \( p, q \) are in stack) as 1.
Pass II (Sweep)

- `free_list` is re-constructed

```
h          0     ...     0  null
       p  0   "     0  null
          q
```

- MB is set to 0 everywhere
- After this: `t=new node()` will be processed

Pros and cons

- Pros
  - Reclaims all free blocks
  - Only called into action when heap overflows

- Cons
  - When it’s called upon, everything will stand still
  - Need to do two passes (one for mark, the other for sweep)
Algorithm 3: copy collection

- Heap partitioned into two halves; only one is active.

Copy collection
Fenichel-Yochelson-Cheney (FYC) - 1970

- Activation
  - Triggered by t=new node() and
  - free pointer outside the active half
**FYC copy collection**

- Copy reachable blocks in **from** to **to compactly**
- Leave forwarding address behind
- Flip the roles of from and to

**Comparison**

- Benjamin Zorn (1990)
  - M&S 5% slower and uses 40% less memory than copy collection
- Memory utilization ratio ("residency")
  - $r = \text{# of used blocks} / \text{total # of heap blocks}$
  - If $r << 0.5$ : Copy collection
  - Otherwise: Mark-sweep
Garbage Collection Summary

- Modern algorithms are more elaborate.
  - Most are hybrids/refinements of the above three.
- In Java, garbage collection is built-in.
  - runs as a low-priority thread.
  - Also, System.gc may be called by the program.
- Functional languages have garbage collection built-in.
- C/C++ default garbage collection to the programmer.