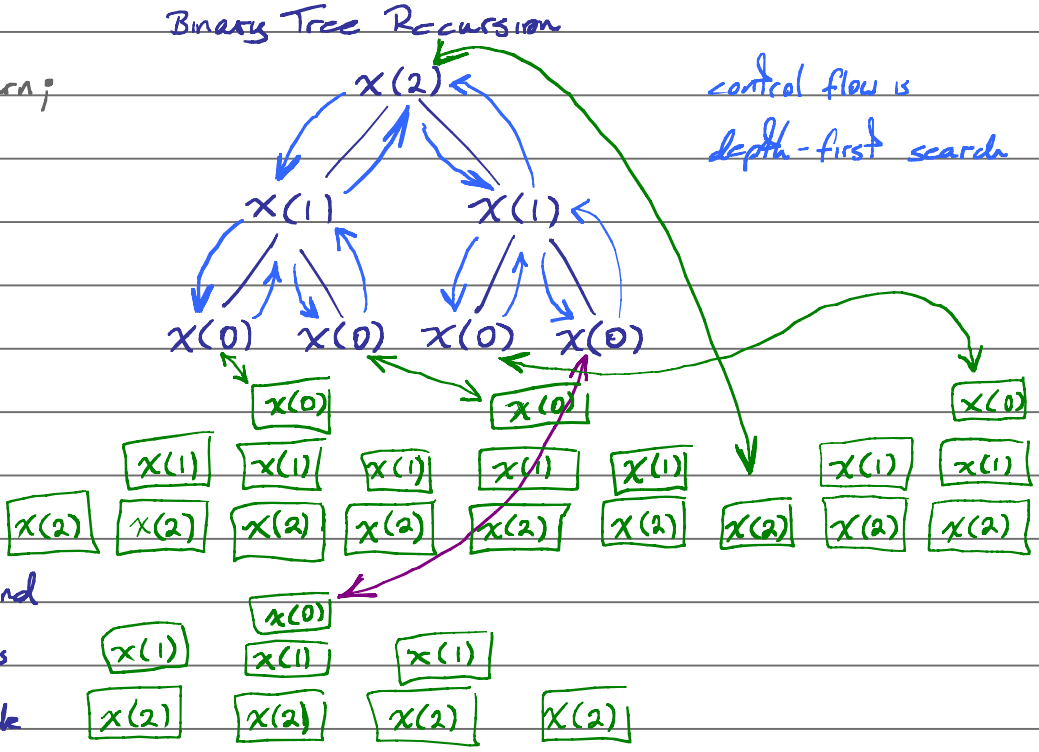
3/6 11-03-09

Note Title 11/3/2009

An example of tree recursion:

```
void (int n)
{
  if (n == 0) return;
  else
  {
    x(n-1);
    x(n-1);
    return;
  }
}
```



- The peaks in the runtime stacks correspond to the deepest leaf nodes
- The # of A.R.s at peak size is approximately equal to the height of the tree of function calls

Memory Mapping and Memory Management

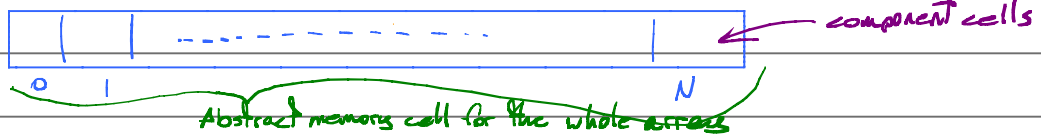
- Arrays
- Class Objects
  - Dynamic binding of method functions, inheritance, polymorphism
- Heap memory for dynamically created data objects.
- Garbage collection: automatically reclaims data objects not used.

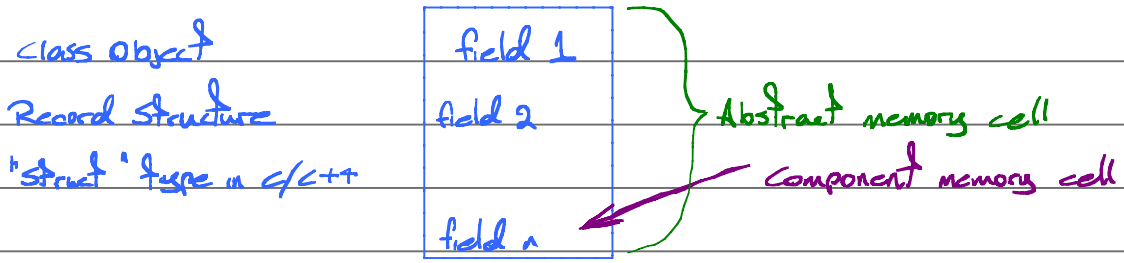
Abstract Memory Cells = • regions of real/virtual main memory that holds well-defined data values/objects.

• an abstract memory cell is compound if it consists of component abstract memory cells

ex:

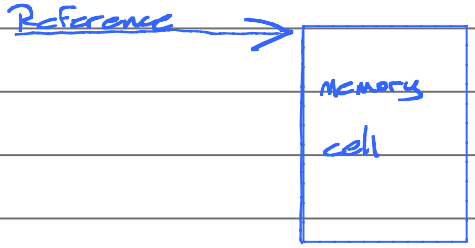
Arrays



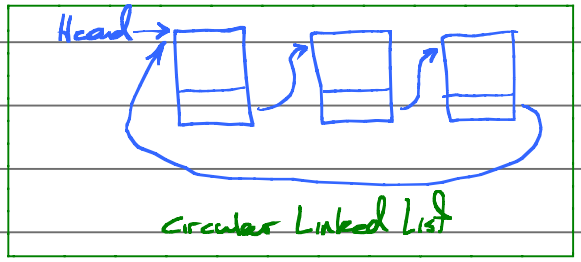


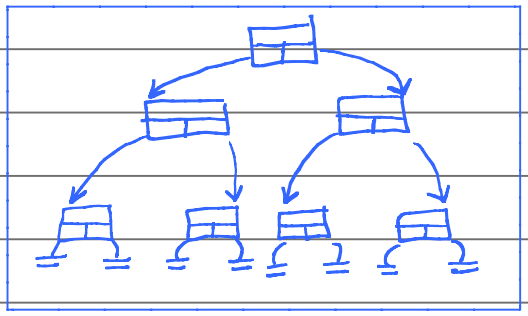
Reference / Pointer / Link

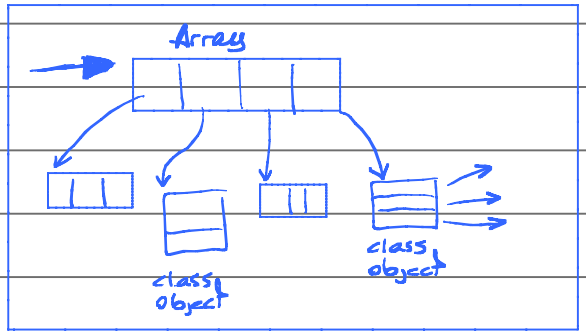
Physically, a reference (pointer, link) is the address of an abstract memory cell



References are crucial for dynamic creation of linked data structures







Memory Mapping OF Arrays

- $[a_1 \ldots b_1, \ldots, a_n \ldots b_n]$ denotes an anonymous n-dimensional array
- $a_i \leq b_i$ for each $1 \leq i \leq n$
- The k^{th} index ranges from a_k through b_k
- $[i_1, \ldots i_n]$ denotes $[i_1, \ldots i_n]^{th}$ element $(a_k \leq i_k \leq b_k$ for each $k : 1 \leq k \leq n)$

Contiguous Allocation

- n-dimensional $\xrightarrow{\text{how to map}}$ linear memory
- generalized row-major order $[a_1 \ldots b_1, \ldots, a_n \ldots b_n]$ is structured as 1-dimensional array $[a_1 \ldots b_1]$ of elements which are each (n-1)-dimensional

array $[a_2 \dots b_2, \dots, a_n \dots b_n]$. Recursively, $[a_2 \dots b_2, \dots, a_n \dots b_n]$ is structured as a 1 dimensional array $[a_2 \dots b_2]$ of elements which are each $(n-2)$-dimensional arrays $[a_3 \dots b_3, \dots, a_n \dots b_n]$, and so on.

$[0 \dots 1, 0 \dots 1, 0 \dots 1]$

Row-Major	Column-Major
0 0 0	0 0 0
0 0 1	1 0 0
0 1 0	0 1 0
0 1 1	1 1 0
1 0 0	0 0 1
1 0 1	1 0 1
1 1 0	0 1 1
1 1 1	1 1 1

Symmetry Law = $[i_1, \dots i_n]$ - element of $[a_1 \dots b_1, \dots, a_n \dots b_n]$ in row-major order is in the same position as $[i_n, \dots i_1]$-element of $[a_n, \dots b_n, \dots, a_1 \dots b_1]$ in column-major order

BA: Base Address = the start address of the memory cell allocated to $[a_1, \dots, a_n]$

ES: Element Size = the size of each memory cell allocated to array elements, measured by the # of addressable memory units (e.g. bytes)

$$\text{Address}([i_1, \dots, i_n]) = BA + \text{Rank}([i_1, \dots, i_n]) \times ES$$

where $\text{rank}([i_1, \dots, i_n])$ = the # of elements preceding $[i_1, \dots, i_n]$

1-dimensional $[a_1 \dots b_1]$ = no difference between row-major and column major

$\text{rank}([i_1]) = i_1 - a_1$

$\text{address}([i_1]) = BA + (i_1 - a_1) \times ES$

$= \underbrace{BA - a_1 \times ES}_{\text{virtual base address (virtual origin)}} + i_1 \times ES$