Memory Management

Based on the slides of Maurizio Gabbrielli

Types of memory allocation

- The life of an object corresponds (typically) with three memory allocation mechanisms:
  - **Static**: Memory allocated at build time (e.g., global variables)
  - **Dynamic**: Memory allocated at run time
    - Stack:
      - Objects allocated with LIFO policy
    - Heap:
      - Objects allocated and deallocated at any time (pointers)

Static Allocation

- An object has an absolute address that is maintained for all the execution of the program
- They are usually statically allocated:
  - Global variables
  - Local Variables of subprograms (without recursion)
  - Constants that are known at build time
  - Tables used by run-time support (for type checking, garbage collection, ...)
- Often used memory protected areas

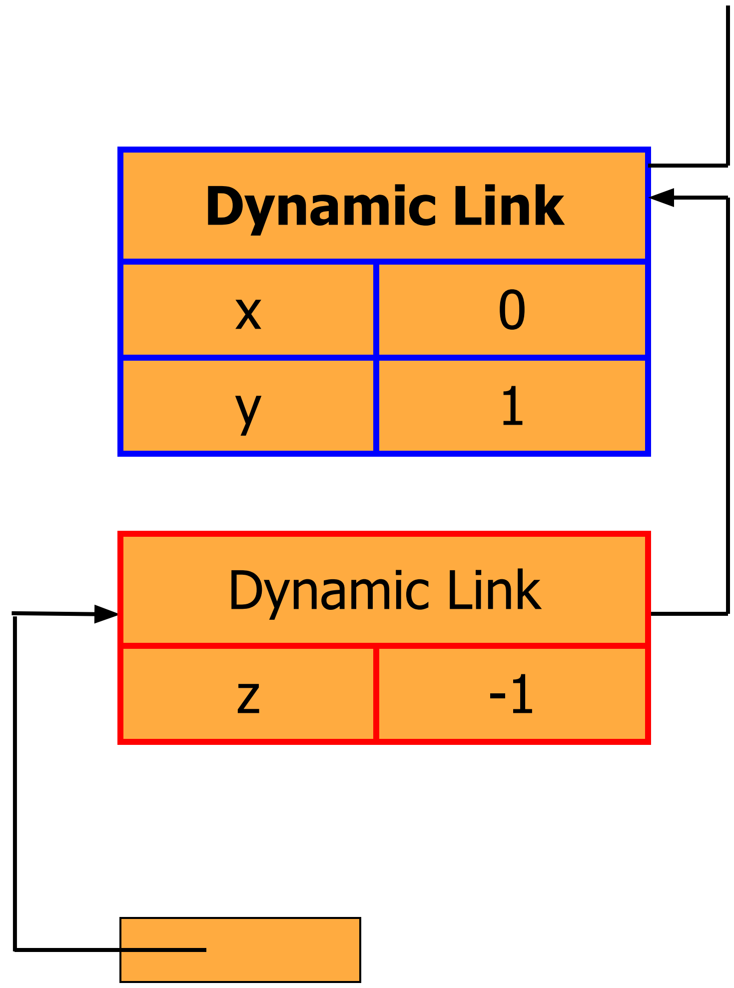
Dynamic allocation: Stack

- With **recursion the static allocation is not enough**:
  - A run time can exist multiple instances of the same local variable as a procedure
- Each instance of a run-time sub-program has a portion of memory called **Activation Record** (or frame) containing information about the specific instance (return address)
- Similarly, each block has its own activation record
- The **stack** (LIFO) is the natural data structure to manage activation records because (recursive) procedure calls and blocks are nested one in the other

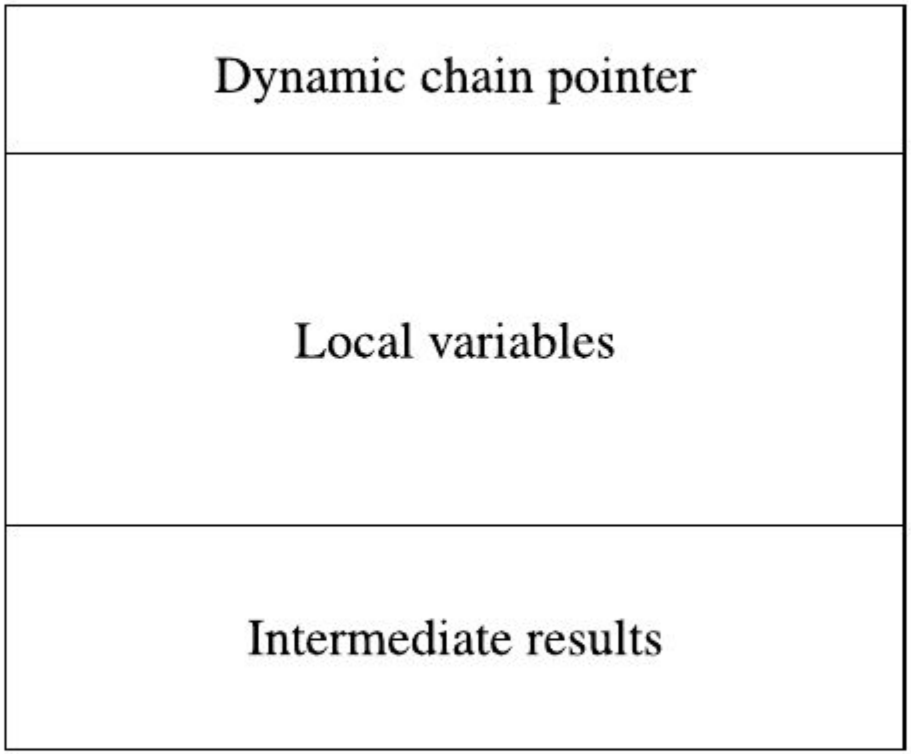
Example

```
{int x = 0;
 int y = x + 1;
        {int z = (x + y) * (x-y);
        };
};
```

Push record with space for x, y
Set values of x, y
    Push record internal block
    Set value for z
    Pop record for internal block
Pop Record for external block



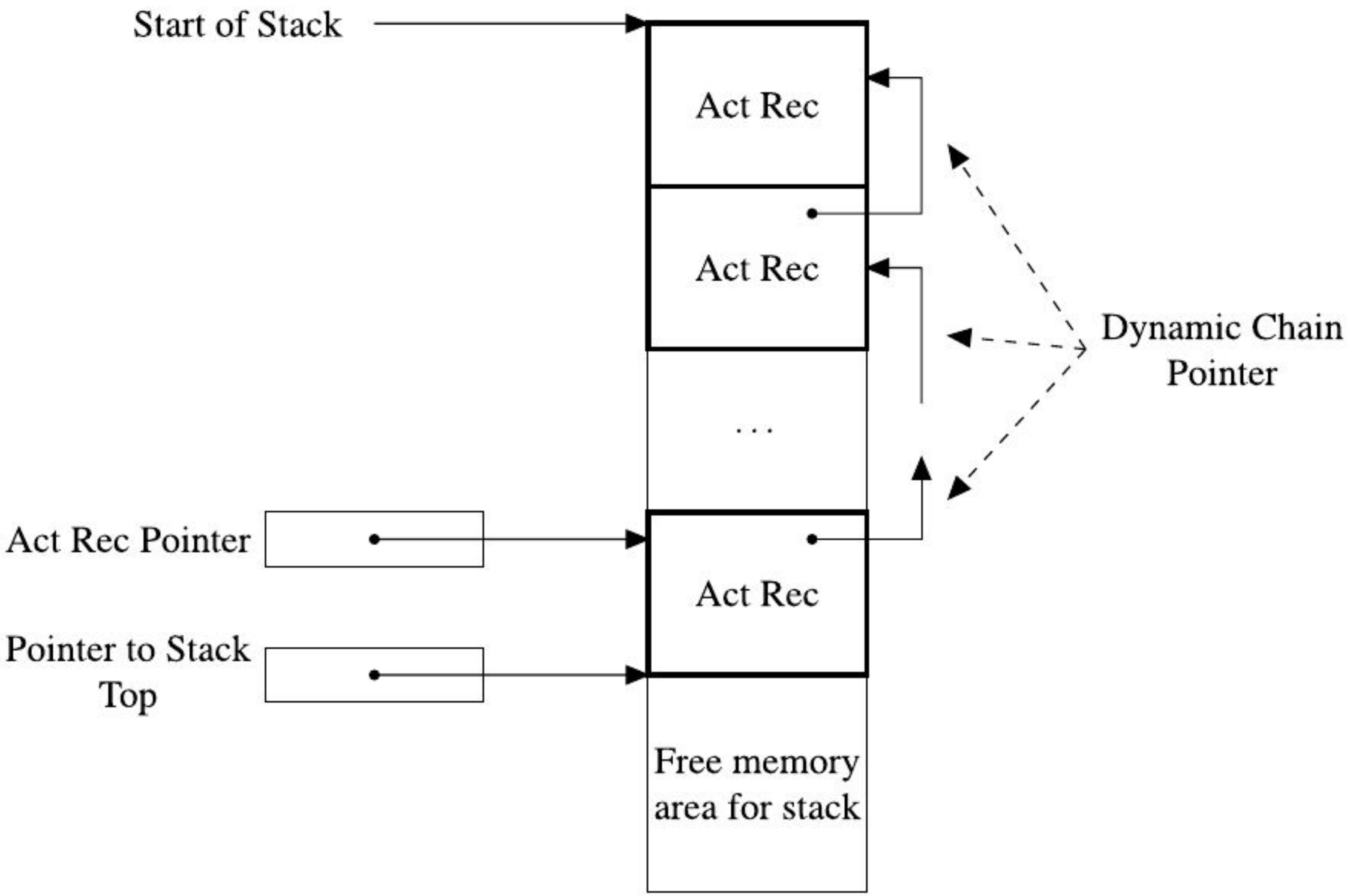
Activation Record for anonymous blocks



Activation Record for procedures

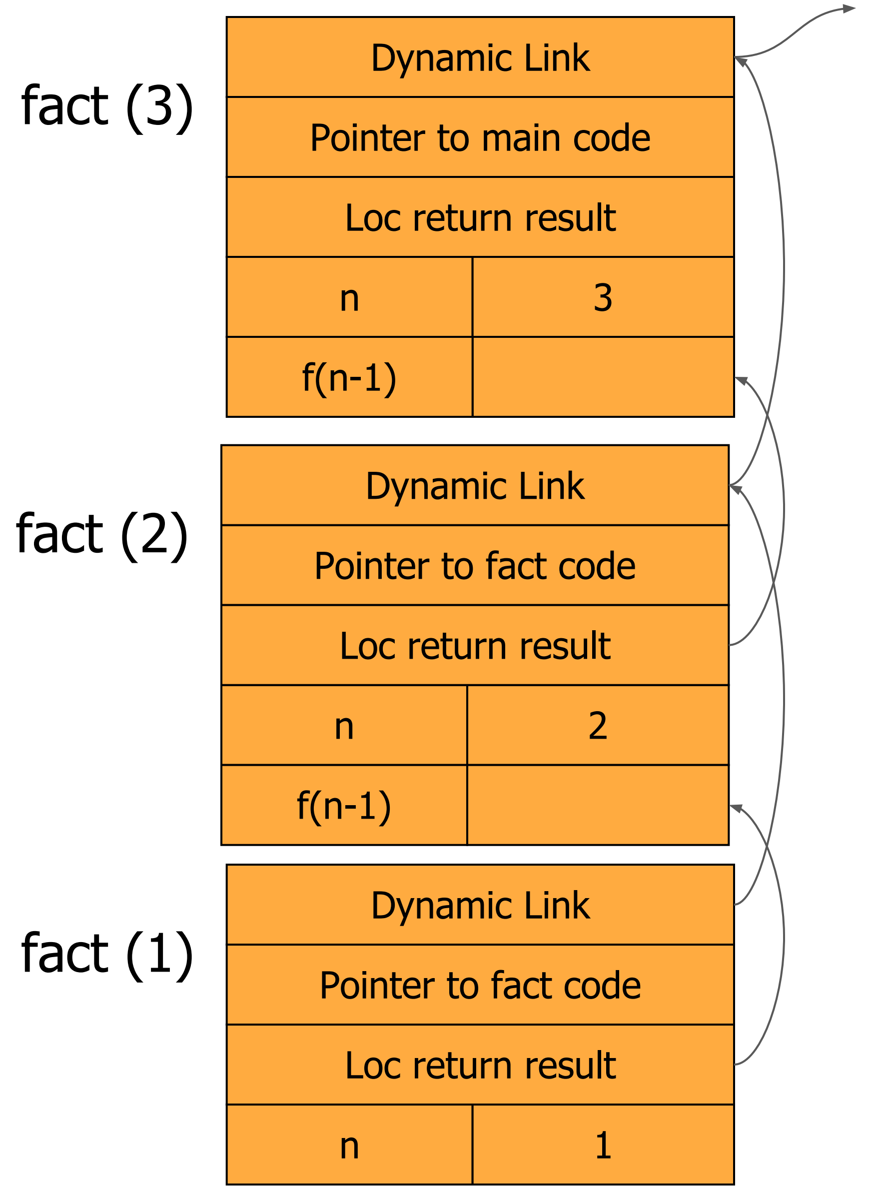
Dynamic Chain Pointer
Static Chain Pointer
Return Address
Address for Result
Parameters
Local Variables
Intermediate Results

Stack Management



Example

```
int fact (int n) {
    if (n < = 1) return 1;
    else return n * fact(n-1);
 }
```



Stack management: entering into the block

- Calling sequence and prologue perform the following tasks:
  - Changing the program counter
  - AR allocation on stack and updating pointers
  - Passing parameters
  - Saving registers
  - Possible initializations
  - Transferring control

Stack Management: exiting the block

- Return values from the call to the caller, or the value computed by the function
- Restoring registers
  - In particular, the old value of the pointer to the AR must be restored
- Possible finalization
- Deallocating space on the stack
- Restoring the program counter value

Dynamic allocation with heap

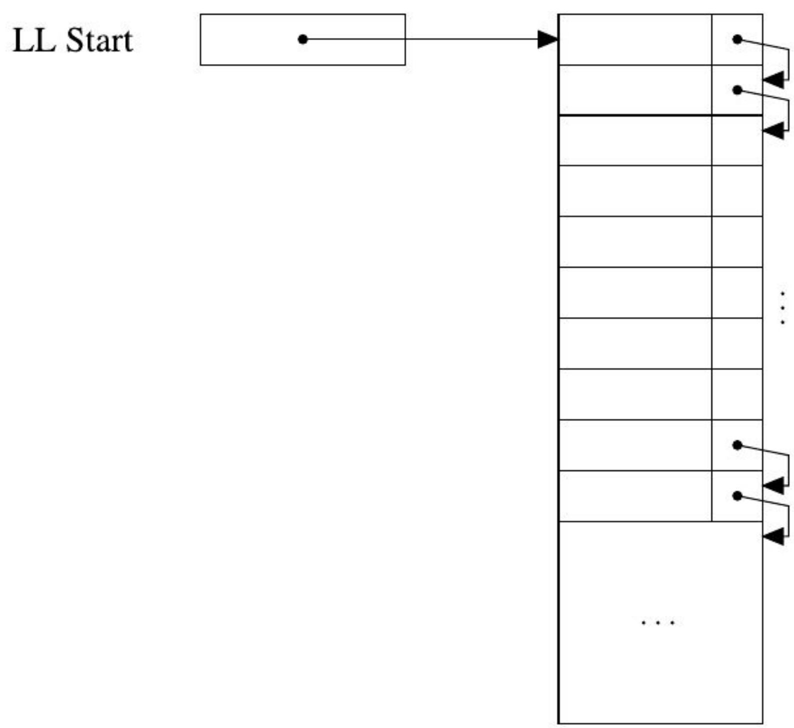
- Heap: memory region whose (sub) blocks can be allocated and deallocated at arbitrary times
- **Necessary** when the language allows
  - **Explicit allocation of memory** at run-time (e.g. dynamic data structures and pointers such as lists, trees....)
  - **Variable-size objects** (strings, collections...)
  - Objects whose life does not have a pre-defined duration (i.e., no FIFO life)

Dynamic allocation with heap

- Heap management is not trivial
  - Efficient space management: fragmentation
  - Access speed
- Two possibilities:
  - languages that allow only fixed size blocks to be allocated
  - languages that allow variable size blocks to be allocated

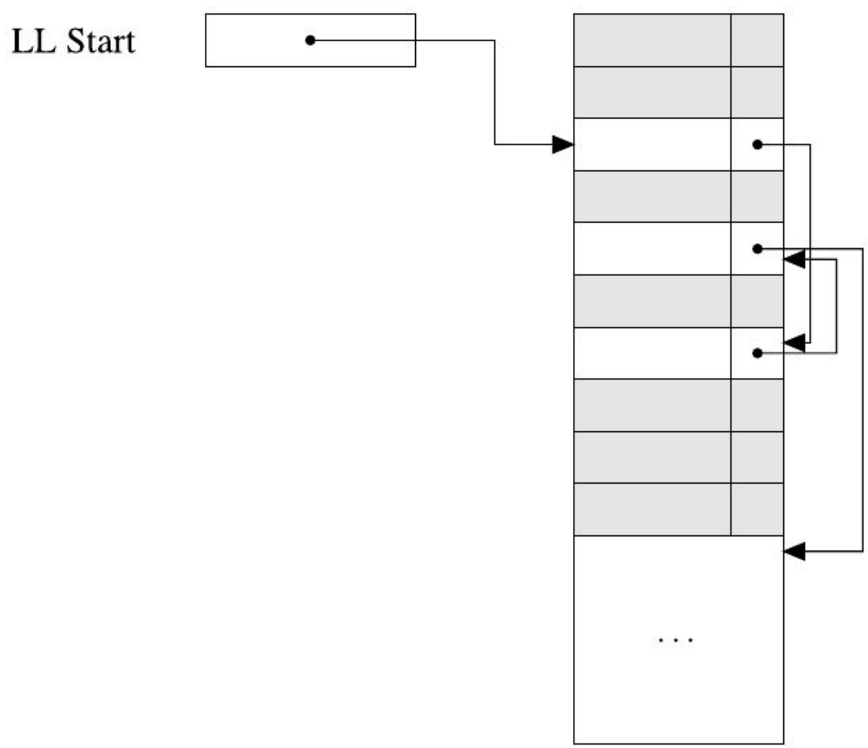
HEAP: **Fixed** size blocks requests

- Heap divided into fixed size blocks
- Originally: all connected blocks in the free list



HEAP: Fixed size blocks

- Allocating: get one block, remove from free list
- Deallocation: returning to free list

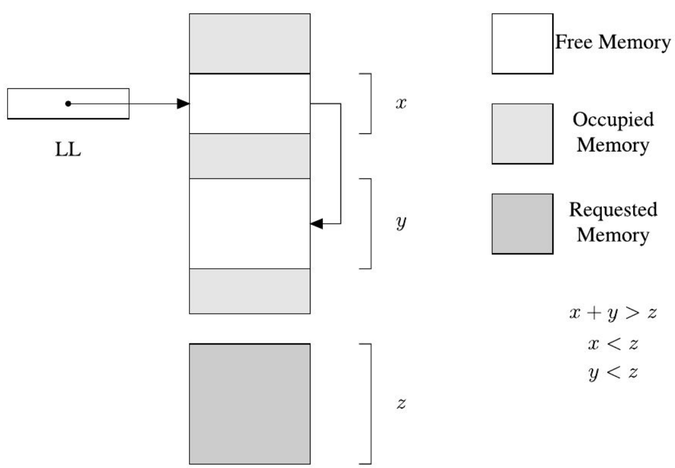


HEAP: **Variable** size blocks requests

- Start with only one block of memory
- For each allocation request: search appropriate block size
  - First Fit: first block big enough
  - Best Fit: the smaller size, large enough
- If the chosen block is much larger than it is needed, it is divided into two and the unused part is added to the FL
- When a block is de-allocated, it is returned to the FL (if an adjacent block is free, the two blocks are "merged" into a single block).

Fragmentation

- Internal Fragmentation
  - the space required is X,
    - a block of dimension Y > X is allocated.
    - Y-X space is wasted
- External Fragmentation: space needed but it is unusable because it is divided into too small scattered pieces



Heap Management

- First fit or Best fit with variable size:
  – First fit: Faster, worse memory occupancy
  – Best fit: Slower, better memory occupancy

- With only one free list for fix size the cost allocation anyway linear in the number of free blocks. To improve:
  – Keep multiple free lists

Multiple free lists

- Multiple free lists, for blocks of different sizes
- The breakdown of the blocks between the various lists can be
  - Buddy system: k lists; the k list has blocks of size 2^k
    - if request allocation of block of 2^k but size not available, block 2^{k+1} divided into 2
    - If a block of 2^k is de-allocated is pooled to his other half (*Buddy*), if available
  - Fibonacci numbers instead of powers of 2 (grow slower)