Dynamic storage allocation

Stack allocation - LIFO (last-in first out)
  - Array and pointer

Allocate x bytes:
  \[ q = p; \]
  \[ p += x; \]
  return q;

Free:
  \[ p = q; \]

Issues:
  - Inline (esp. for small allocations)
  - May want to round up to word (or cache block) boundary
  - Must free consistently with stack discipline => limited applicability, but fast when it works.

Can sometimes use call stack, e.g., `alloca()`. `free()` is deprecated. (Hard to control stack size in some environments, and compiler is more efficient with fixed-size function activation records (frames)).

Heap allocation « Not heap data structure »

C: `malloc()` and `free()`
C++: `new` and `delete`

Unlike Java and Python, C/C++ has no garbage collector. Storage that is allocated must be freed explicitly by programmer. Otherwise, memory leak! Also, watch for dangling pointers and double frees.
Fixed-size allocation

Allocation: remove head of free list - O(1) time
Free: add block to free list - O(1) time

Few ops + good temporal locality

Problem: External fragmentation: blocks distributed across memory \Rightarrow loss of spatial locality

Variable-size allocation

Idea: Leverage efficiency of fixed-size allocation. Accept some internal fragmentation

Binned free lists.

- Allocator supports a reduced set of canonical sizes, typically \$2, 2, 4, 8, 16, \ldots 3\$
- (or slightly larger - bookkeeping info, cache alignment)

Allocate \( x \):
- Look in bin \( \lceil x / 7 \rceil \),
- If empty, find block in larger bin and split.
- No larger blocks, allocate new address space.
Typical storage layout of program:

- high addr
- stack
- dynamically allocated
- heap
- uninitialized data (lost)
- initialized data (read from program file)
- code

Q. 64-bit virtual memory ⇒ never run out of address space. Why not just allocate what you need and never free?
A. VM is backed up by physical memory, e.g., disk. Also, severe external fragmentation.

Goal: use as little VM as possible.

**Theorem.** Let $M$ be the maximum amount of memory in use at any time by a program. Then a bin free list uses at most $O(M \log M)$ VM address space.

**Proof.** No bin ever contains more than $2M$ memory. At most $\log(2M)$ bins. □
Theorem Let $M$ be the VM address space used by the optimal allocator (with no coalescing). Then binned free lists uses at most $6M$ VM addr. space.

Coalescing:
- Splice adjacent blocks together to make larger block.
- Ameliorates external fragmentation.
- No good bounds on effectiveness.
- Works in practice, because storage tends to be deallocated as a stack (LIFO) or in batches.
- Clever schemes for finding adjacent blocks, e.g., "buddy" system.
Garbage collection

Idea: Rather than user freeing storage, runtime does it. Built-in (Java, Python) (or build by hand).

Roots: storage directly accessible (global vars, stack). Objects not reachable from roots through pointers can be recycled.

Key issues:

- Identify pointers in objects (strong typing)
- Prohibit pointer arithmetic - may slow down some codes.

Reference counting

Keep count of # pointers to each object. If pointer count drops to 0, put in free list.

Problem: Cycle never GC'ed. Nevertheless, good scheme for acyclic structures.
Abstraction

Objects form a directed graph.
GC objects not reachable from roots.
=> graph searching (e.g., DFS or BFS).

Recall BFS:
Mark roots $r$ and put into FIFO $Q$;
while ($Q \neq \emptyset$)
    $u \leftarrow$ dequeue ($Q$);
    for (each ($u, v$) incident on $u$)
        if $v$ unmarked
            mark $v$;
            enqueue ($Q$, $v$);

Checkout

Observe: All reachable objects are placed in contiguous storage in $Q$.

Copying GC:

When From space is full, copy to To space.
Issue: Since To addr ≠ From addr, ptrs must be updated.

1. When obj. copied to To space, store forwarding ptr in From obj, which implicitly marks it as moved.
2. When obj. removed from Q in To space, update all its pointers.

Ex. Before

From

To

Ex. After

From

To

• Linear time
• After copying, allocate new unused space equal to (or const fraction of) used space
  \[ \Rightarrow \text{amortize GC time, keep VM addr space small.} \]

Old To becomes new From
New To allocated at start if room; otherwise after new From
\[ \Rightarrow \text{VM addr space} \leq \text{const. optimal} \]

Exercise: What is constant?

Other GC strategies
• Mark and sweep
• Generational