Topic 18 –Memory Management

Key Ideas

- system stack vs. heap
- automatic vs. manual memory management
- fragmentation
- first fit, best fit, worst fit
- the 50% rule
- dlmalloc
- garbage collection
- mark and sweep
- reference counting
- compaction
- copying

Code Gen for New and Delete

Rules in WLP4 that Deal with Arrays and Pointers

- factor → NEW INT LBRACK expr RBRACK

```
code(factor) = code(expr)     ; calc size of array requested
        add $1, $3, $0        ; move size to $1
        new($1)               ; call new
```

- statement → DELETE LBRACK RBRACK expr SEMI

```
code(stmt) = code(expr)       ; calc addr of array to delete
        add $1, $3, $0        ; move address to $1
        delete($1)            ; call delete
```

- to use dynamic memory we need to initialize data structures in the alloc.merl library

```
wain_prolog:                  ; only initialize once, if wain has an
       init($2)               ; array as a param, size is in $2
```

Overview of Memory Management

The Challenge

- Procedure arguments, return values, and local variables can all be handled quiet elegantly with the system stack.
- Nested procedure calls and returns values follow a first in last out pattern just like pushing and popping from a stack.
- Dealing with commands like **`new`** and **`delete`** (i.e. *dynamic allocation* and reclamation) is a much more problematic issue.
- *The Problem:* **`new`** and **`delete`** can be called in an unpredictable pattern (e.g. in an if statement)
  - they don't necessarily follow any nice structure like First In First Out or Last In First Out
  - e.g. if new was called in the order: **`new a; new b; new c;`** **`a`**, **`b`** and **`c`** could be deleted in any order.

Overview of Memory Management

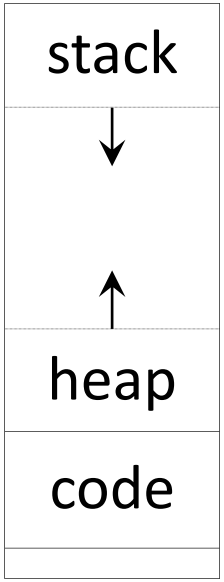
The Challenge

- *Key differences:* Local variables disappear once the function that they are declared in returns, but dynamically allocated arrays can remain even after the function has returned.
- Many data structures can grow and shrink dynamically (e.g. a linked list), i.e. their size is not known at compile time
- *Consequences:* Because of these differences, it is not efficient to store dynamically allocated memory in the system stack.
- *Solution:* Instead another region of memory is reserved for dynamically allocated memory: the *heap*
- Here heap means RAM available for dynamic allocation not a balanced binary tree (as in CS240/SE240).

Overview of Memory Management

Solution: Typical Layout in Memory

- The *stack* is located at the high end of memory (i.e. large values for addresses) and grows down.
- The *heap* is located just above the code and grows up.
- The *code* is located near the low end of memory and it does not change size as the program runs.
- During the running of a program
  - the size of the stack can increase or decease as functions get called and return
  - the size of the heap can increase and decrease as functions call new and delete



Overview of Memory Management

The Components

- When looking at memory management there are three tasks to consider...

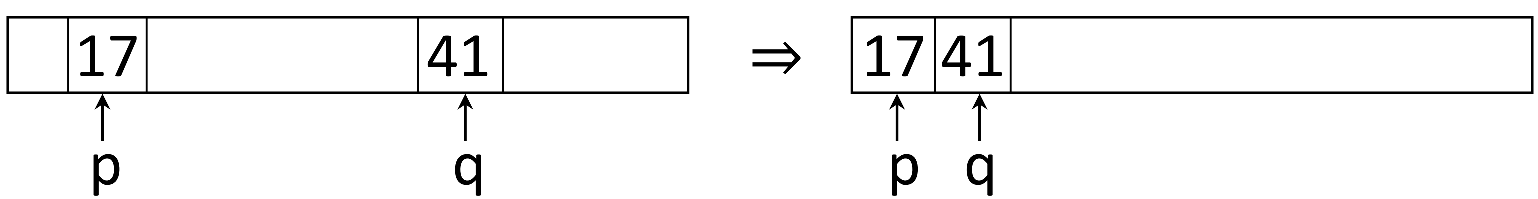
	System Stack	Heap
1. Initialization	done by O/S	init
2. Allocation	push()	new
3. Reclamation	pop()	delete

- The operating system (O/S) initializes the system stack.
- Procedures are implemented to manage the allocation and reclamation of the system stack efficiently.
- How the heap is managed varies: there are many possibilities ...

Heap Allocation and Deallocation

Varieties of Memory Management

- memory can be *allocated* implicitly (it just happens) or explicitly (i.e. the function **`new`** is called).
- memory can be *reclaimed* implicitly (it just happens) or explicitly (i.e. the function **`delete`** is called).
- memory can be *allocated* in one size only (a *fixed size*) or in many sizes (a *variable size*)
- some languages (not WLP4 or C++) allow pointers to be *relocated* in order to fill in spaces between allocated memory



Overview of Memory Management

Implicit vs. Explicit

- Many languages (like Java and Python) have implicit / *automatic memory management*
  - the program creates new objects and a procedure runs in the background that decides when to free up the memory for the object because it is no longer being used (a.k.a. garbage collection).
- Other languages (like WLP4, C, and C++) have explicit / *manual memory management*
  - the programmer calls **`delete[]`** on any memory that is no longer needed
  - the risk is you can call **`delete`** too early, too late or too often

Overview of Memory Management

Pros of Automatic Memory Management

With automatic memory management you avoid or substantially reduce

- *dangling pointer* errors: using memory that has been freed, i.e. you have called **delete** *too early*

```
int* ia = NULL;
ia = new int[100];
delete [] ia;
  ⋮
ia[0] = 17;      // error: dangling pointer!
```

- risks: if that memory location is being used by another data structure, you are unintentionally modifying it in an unpredictable way

Overview of Memory Management

Pros of Automatic Memory Management

With automatic memory management you avoid or substantially reduce

- *memory leaks:* you allocate memory but then have no pointers pointing to it, i.e. you have called **`delete`** *too late*

```
int* ia = NULL;
ia = new int[100];
 ⋮
ia = NULL;        // error: access to memory is lost!
```

- the program slowly uses up more and more memory
- risks: memory exhaustion (i.e. running out of memory)
- the risk increases if the program runs for a long time

Overview of Memory Management

Pros of Automatic Memory Management

With automatic memory management you avoid or substantially reduce

- *deleting twice:* you call **`delete[]`** on the same memory location multiple times, i.e. you have called **`delete`** *too often.*

```
int* ia = NULL;
ia = new int[100];
delete [] ia;
  ⋮
delete [] ia;    // error: freeing twice!
```

- risks: can crash the system

Overview of Memory Management

Cons of Automatic Memory Management

With automatic memory management you

- use more resources (i.e. time to track memory usage)
- may have a performance impact
- possible stalls in program execution (i.e. not good for some real time programming applications)

Manual and Automatic Memory Management Commonalities

- With both you still need to track which locations in RAM are
  - *being used*
  - *free* (available for use)

Overview of Memory Management

Key Observations

- You need to *carve out an arena* from somewhere, i.e. a large contiguous area of memory that gets allocated once and then is handed out in pieces using calls such as **`new`** or **`malloc`**
  - perhaps from the stack during the prolog for wain()
  - or the O/S provides it for you in memory just above the code
  - we call this arena the *heap*
- this arena provides an area of memory that the **`new`** and **`delete`** routines manage
- ***`new`*** (or **`malloc`**) *is easy if you don't have **`delete`*** (or **`free`**) and don't reuse the memory
- however you could quickly run out of available memory if it is not reused

Overview of Memory Management

Key Observations

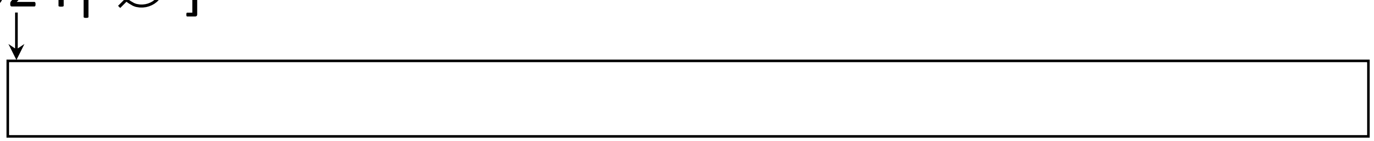
- The function *free can be implemented with an available space list*
  - easy if the allocations are a fixed size
  - difficult if allocations are variable-sized because you have to find the a suitable-sized hole
- if heap is only ½ full, you'll find a hole fairly quickly

Version 1: List of Free Blocks

Idea and Initialization

Features: variable sized block, explicit allocation, explicit reclamation, no pointer reallocation.

- idea: create a linked list of free blocks O(1)
- *init:* initially the entire heap is free and the linked list contains one entry (say 1024 bytes)
- free →[1024| ∅]



Version 1: List of Free Blocks

Allocation

- if 16 bytes are requested
  - allocate 20 bytes:
    - the 1st part (4 bytes) stores the size of the block
    - the 2nd part (16 bytes) stores the data
  - return a pointer to the start of the data portion

    16 |

    ↑

    p

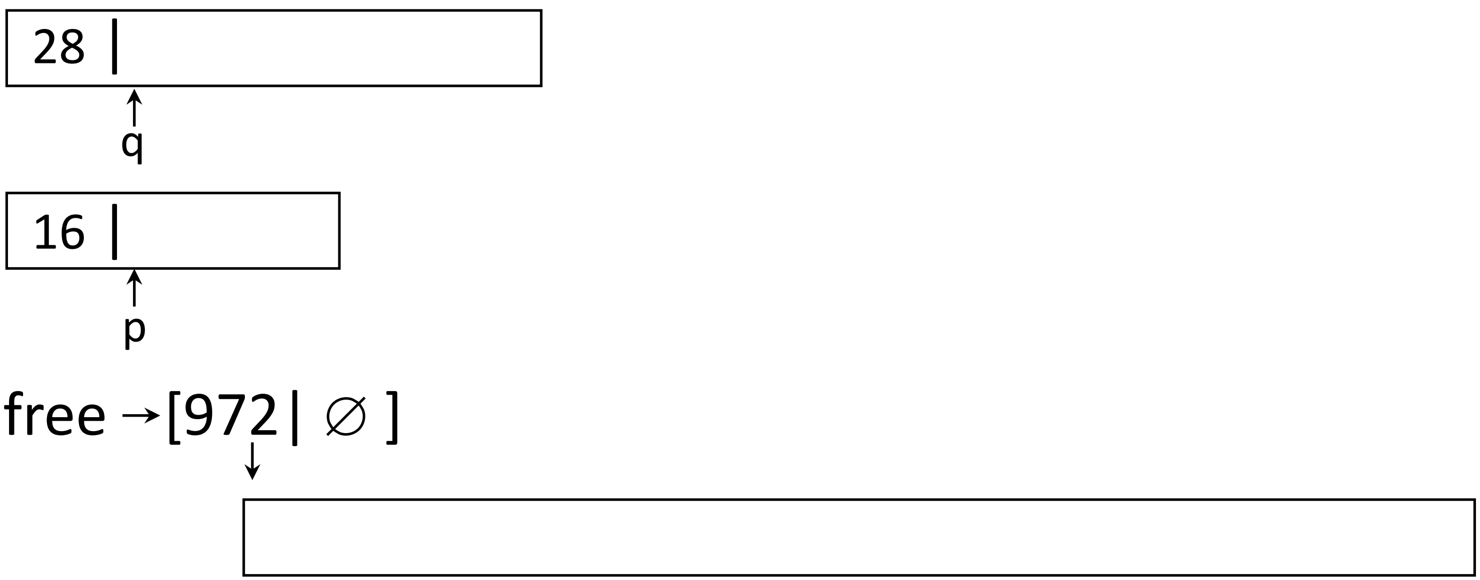
  - the free list now contains 1004 bytes

    free → [1004 | ∅]

Version 1: List of Free Blocks

Allocation

- if 28 bytes are requested next
  - allocate 32 bytes, store the size in the 1st part and return a pointer to the start of the 2nd part

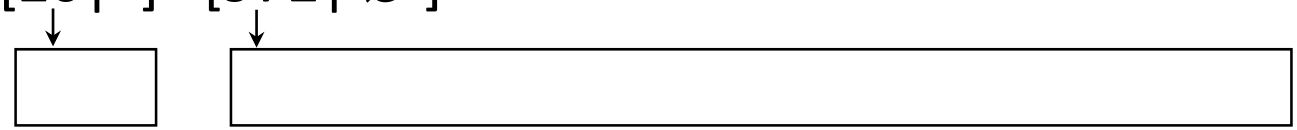


Version 1: List of Free Blocks

Reclamation

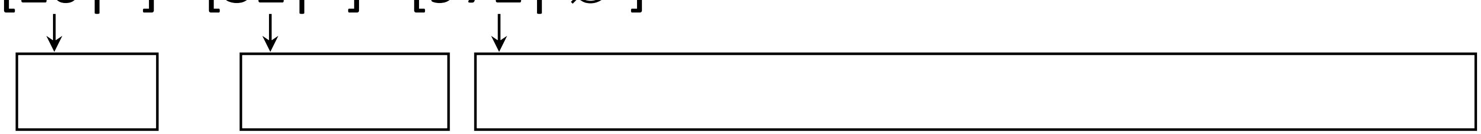
- suppose the first block is freed, i.e. **`delete [] p`**;
  - **`delete`** checks p[-1] to determine how much memory has been freed and adds it to the free list

free → [20 | •] → [972 | ∅]



- suppose the second block is freed, i.e. **`delete [] q`**;
  - **`delete`** checks q[-1] to determine how much memory has been freed and adds it to the free list

free → [20 | •] → [32 | •] → [972 | ∅]

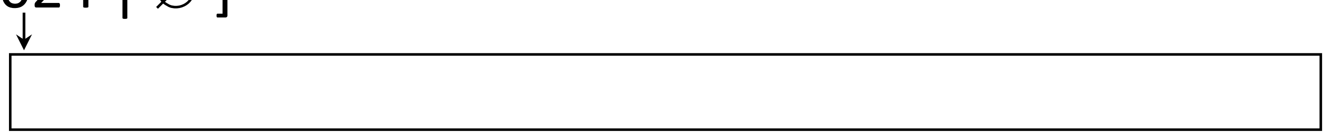


Version 1: List of Free Blocks

Reclamation

- the system will recognize that these blocks are adjacent in RAM and merge them together

free→ [1024 | ∅]



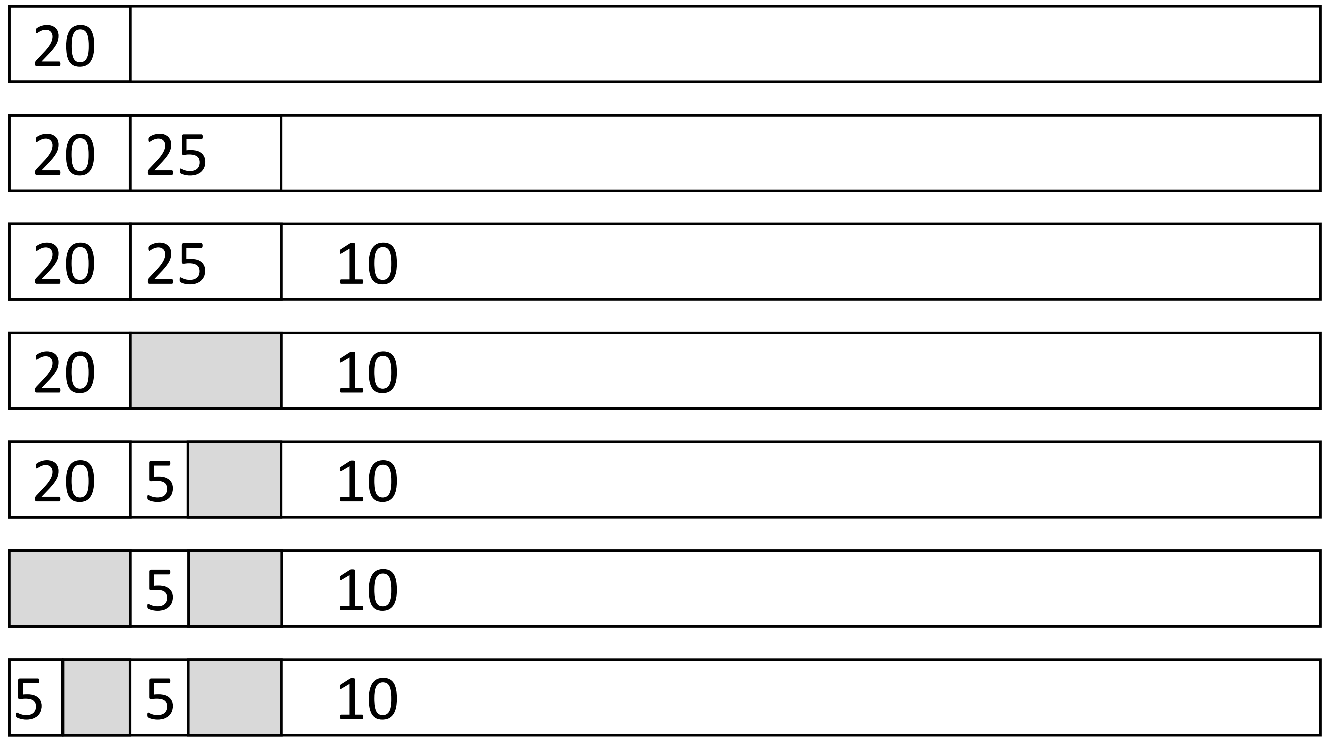
Fragmentation

- *Problem:* repeated allocation and reclamation can create gaps in the heap
- called *fragmentation*, i.e. even though there are *n* bytes free in the heap, you *may not be able to allocate a block of n contiguous bytes*

Version 1: List of Free Blocks

Fragmentation

- alloc 20
- alloc 25
- alloc 10
- free 25
- alloc 5
- free 20
- alloc 5



- *Idea:* to reduce fragmentation don't always choose the first block of RAM big enough to satisfy the request

Version 1: List of Free Blocks

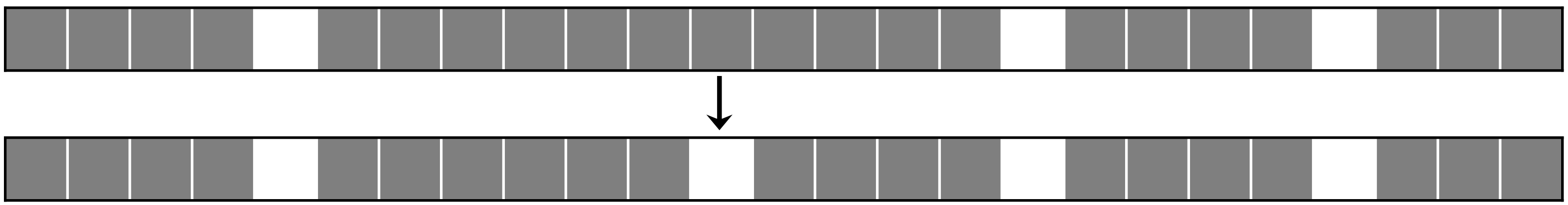
Allocation Strategies

- *first fit:* find the first hole it fits in
  - fast to find location
  - risk of more wasted RAM
- *best fit:* find the location has the least amount of leftover space
  - slow: must search through available RAM
  - less wasted RAM
- *worst fit:* pick the biggest hole, so that you have a relative large hole remains, which can easily satisfy another request
  - slow: must search through available RAM
  - less wasted RAM
- *Moral:* you get what you pay for (in terms of time)

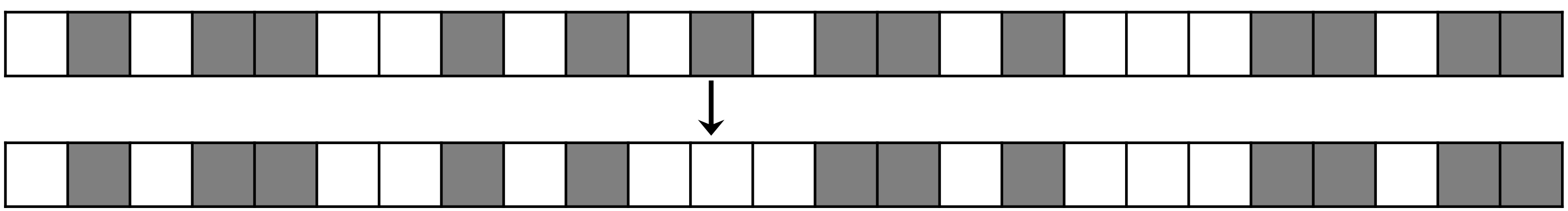
Overview of Memory Management

Key Observations: the 50% rule

- If the heap is relatively full (i.e. there are few holes) then deleting will very likely introduce new holes.



- If the heap is relatively empty (i.e. there are many holes) then deleting will very likely coalesce (i.e. reduce the # of) holes.



- *50% rule:* on average, for first fit, if *n* blocks are allocated then 0.50*n* has been lost to fragmentation (i.e. ⅓ of the total)

Version 2: dlmalloc

Allocation Strategies

- named after its creator, Douglas Lea
- used in C since 1987 (with modifications to allow for multithreaded code)
- *key idea:* distinguish between small allocations, called *smallbin requests* (512 bytes or less), medium (typically 513B to 256KB or less) and large sized requests (greater than 256KB)
- smallbin requests have bins of various sizes, all multiples of 16 starting at 32 bytes, i.e. 32, 48, 64, 80, … 512
- *key idea:* have multiple free lists for holes of different sizes
- medium and large sizes have more sophisticated data structures like tries

Version 2: dlmalloc

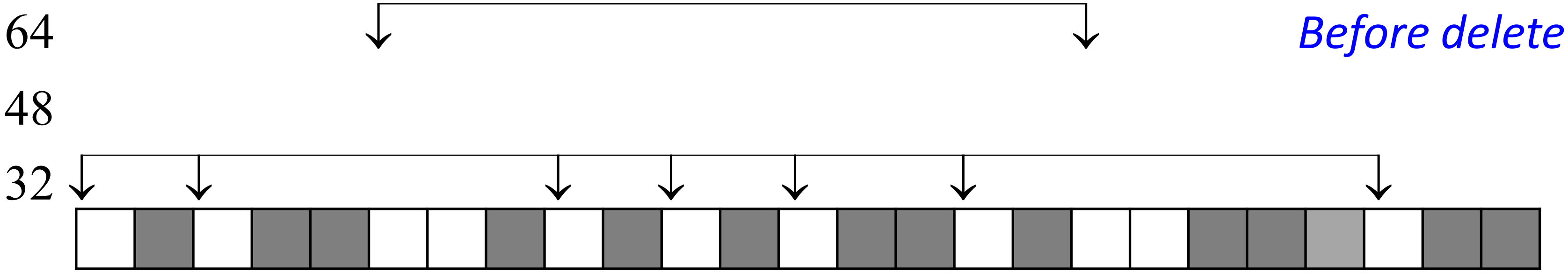
Allocation and Deallocation Strategies

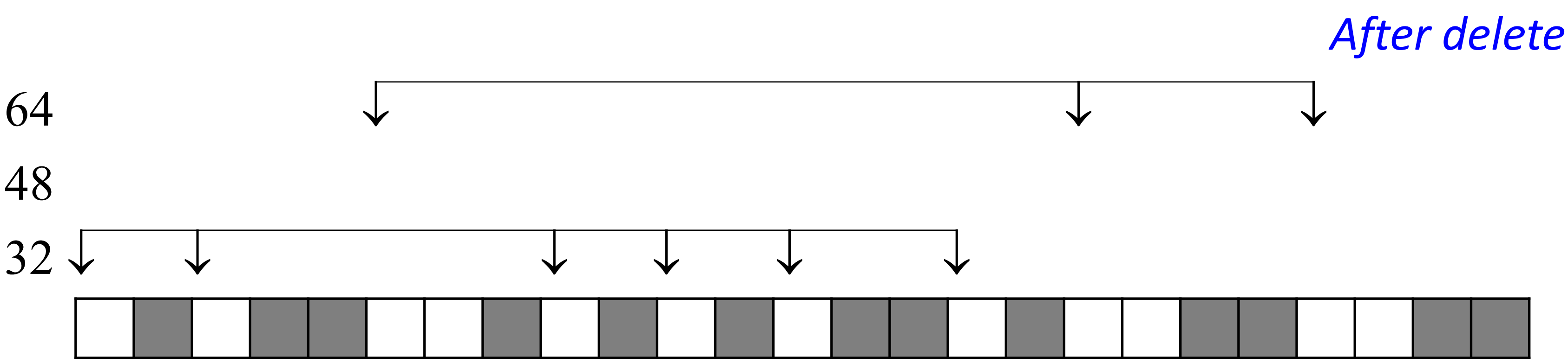
- for small bin requests, each pointer tracks 9 bytes of info, its status (free or inuse) and two copies of it’s size (one at the beginning of the space allocated and one at the end)
- since the lowest bin size is 32, a request for 1 to 23 bytes results in an allocation of 32 bytes because 9 bytes are reserved for bookkeeping
- *when deallocating,* check the neighbour on either side and if the neighbour is free join together to create a larger block
- *when allocating,* if there isn’t a small block available to fulfill a request break up a larger block into smaller ones

Version 2: dlmalloc

Deallocation Strategies

- here the light grey 32-byte block on the RHS becomes free and joins with its neighbour on the right to become a 64-byte block and in the process changes the two free lists





Automatic Memory Management

Recall Manual Memory Management

- used in: C, C++, WPL4
- *the programmer decides* when to allocate and deallocate memory (by a call to new or delete)
- this approach can lead to a number of different types of errors
- just like left and right parenthesis () or braces {}, calls to new and delete must be paired up or it can be a source of error
  - dangling pointer: call delete too early
  - memory leaks: call delete too late (or not at all)
  - deleting twice: call delete too many times

Overview of Memory Management

Pros of Automatic Memory Management

- *The problem:* pointer values can be assigned or changed.

```
1   int* ia = NULL;
2   int* ip = NULL;
3   ia = new int[100];
4   ...                          // What happened here?
5   ip = ia;
6   ...                          // What happened here?
7   delete [] ip;
```

- Question: Is line 7 an error?
- Answer: it depends on what happened on lines 4 and 6.
  Was delete called on **`ia`**? Was **`ip[1]`** or **`ia[1]`** accessed after the delete? Was **`ia`**'s or **`ip`**'s value modified? If you did "**`ip = ip+1`**" did you lose the original value of **`ip`**?

Automatic Memory Management

Recall Manual Memory Management

- The compiler cannot tell for sure if it is an error or not because what happens in 4 and 6 could depend on the input.
  - *e.g.* there could statements that say:

```
if (user closes the browser tab) {
    delete [] ia;
}
if (memory low) {
    delete [] ip;
}
```

- *conclusion:* don’t try to detect if new and delete are properly paired up at compile time as pointer values can be assigned (i.e. copied) or modified.

Automatic Memory Management

Approaches

- *challenge:* need to identify all the pointers
- *Solution 1:* monitor memory access and the values of pointers at runtime (e.g. valgrind)
  - slows down the program so should only be used during testing
  - good testing relies on selecting good test cases
  - hard to guaranteed you've caught all errors
- *Solution 2*: decide when to free up memory automatically, typically called *garbage collection*.
  - two basic approaches here
    - search for unused memory and reclaim it: *mark and sweep*
    - search for used memory and reclaim the rest: *copying* or *compacting*

Automatic Memory Management

Approach 1: Mark and Sweep

scan global variables and the entire stack for pointers
for each non-NULL pointer found
  mark the block in the heap that the pointer is referring to
  if the heap object contains pointers
    then follow those pointers as well
scan the heap
  reclaim any blocks not marked
clear all marks

- since we are following pointers to blocks that could contain more pointers we are searching on a graph, e.g. need some sort of graph traversal algorithm (a CS341 topic), e.g. depth first search

Automatic Memory Management

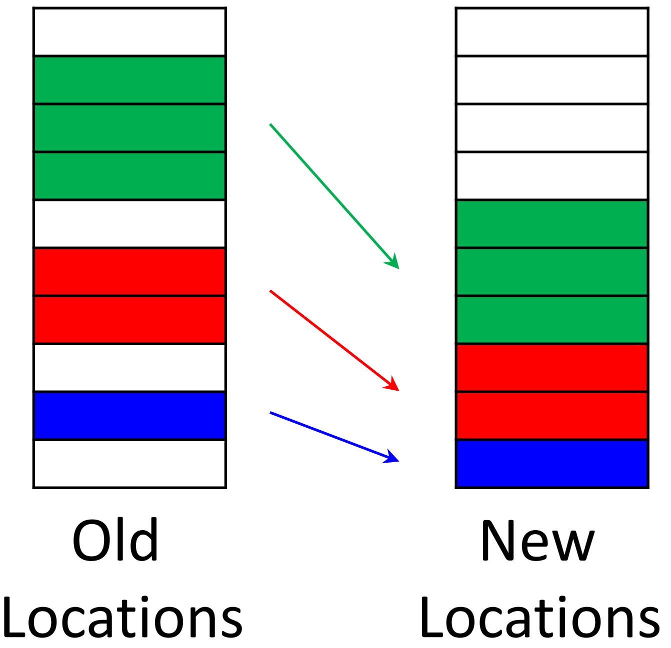
Approach 2: Reference Counting

- for each heap block, keep track of its *reference count,* i.e. the number of pointers that point to it
- this means you must keep track of every pointer and update the references counts each time a pointer is reassigned
- if a block's reference count is 0, then reclaim it
- problem: circular references
  - a pointer in block 1 is pointing to block 2
  - a pointer in block 2 is pointing to block 1
  - if no other pointers are pointing to block 1 or 2 then their reference count is both 1 but collectively they are inaccessible

Automatic Memory Management

Compaction

- after sweep calculate new locations
- in a second sweep, move data (*referents*) and adjust pointer values
  - *pros:* no fragmentation: after the compaction, all reachable data will occupy continuous memory
  - *cons:* not as easy as copying to free locations



Automatic Memory Management

Copying

- heap has two regions: 1) *from* 2) *to*
- allocate only from *from*
- when *from* fills up, all reachable data is copied from *from* to *to* and the roles of *from* and *to* are reversed
  - *pros:* no fragmentation: after the copy, all reachable data will occupy continuous memory
  - *pros:* new and delete are quick
  - *cons:* only half the heap is in use at a time (variants have 3 or 4 regions)
- widely used method