Dynamic Memory Allocation: Advanced Concepts

Andrew Case

Slides adapted from Jinyang Li, Randy Bryant and Dave O’Hallaron

Joke of the day: Why did the programmer quit his job?
Topics

- Implicit free lists
- Explicit free lists
- Segregated free lists
- Garbage collection
- Memory-related perils and pitfalls
Keeping Track of Free Blocks

- **Method 1:** *Implicit free list* using length—links all blocks

- **Method 2:** *Explicit free list* among the free blocks using pointers

- **Method 3:** *Segregated free list*
  - Different free lists for different size classes

- **Method 4:** *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key
Explicit Free list

- **Disadvantage of implicit free list**
  - For each allocation, $O(N)$ blocks are traversed, many of which are not free

- **Explicit free list**
  - Maintain list(s) of free blocks instead of all blocks
  - Need to store forward/back pointers in each free block, not just sizes
**Explicit Free Lists**

<table>
<thead>
<tr>
<th>Allocated block</th>
<th>Free block</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
<td><strong>Size</strong></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

**Payload and padding**

Store next/prev pointers in “payload” of free block.

Does this increase space overhead?
Allocating From Explicit Free Lists

List of free memory

List after allocation

= malloc(…)

Alloca/ng
From
Explicit	
Free	
Lists

List	
of	
free	
memory

List	
after	
allocation
Freeing With Explicit Free Lists

- **Insertion policy:** Where in the free list do you put a newly freed block?
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
    - **Pro:** simple and constant time
    - **Con:** studies suggest fragmentation is worse than address ordered

- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:
    \[
    \text{addr}(\text{prev}) < \text{addr}(\text{curr}) < \text{addr}(\text{next})
    \]
  - **Con:** requires search
  - **Pro:** studies suggest fragmentation is lower than LIFO
Freeing With LIFO: Basic case

- Insert the freed block at root

Check prev block’s footer and next block’s header for allocation status
Splice out prev block, coalesce, and insert new block at root
Explicit List

✓ Allocation is **linear time in # of free blocks** instead of **all** blocks

- Still expensive to find a free block that fits
  - How about keeping different linked lists for different sizes?
Keeping Track of Free Blocks

- **Method 1:** *Implicit list* using length—links all blocks

- **Method 2:** *Explicit list* among the free blocks using pointers

- **Method 3:** *Segregated free list*
  - Different free lists for different size classes

- **Method 4:** *Blocks sorted by size* (not covered)
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key
Segregated List (Seglist) Allocators

- Each *size class* of blocks has its own free list

1-2

3

4

5-8

9-16

... 

- Often have separate classes for each small size
- For larger sizes: One class for each two-power size (buddy-system)
Seglist Allocator

- Given an array of free lists, each one for some size class

- To allocate a block of size $n$:
  - Search appropriate free list for block of size $m > n$
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found

- If no block is found:
  - Request additional heap memory from OS (using `sbrk()`)
  - Allocate block of $n$ bytes from this new memory
  - Place remainder as a single free block in largest size class.
Seglist Allocator (cont.)

■ To free a block:
  ▪ Coalesce and place on appropriate list (optional)

■ Advantages of seglist allocators
  ▪ Higher throughput
    ▪ log time for power-of-two size classes
  ▪ Better memory utilization
    ▪ First-fit of seglist approximates best-fit on entire heap
    ▪ Extreme case: Giving each block its own size class is equivalent to best-fit.
More Info on Allocators

  - The classic reference on dynamic storage allocation

  - Comprehensive survey
  - Available from CS:APP student site (csapp.cs.cmu.edu)
Topics

- Explicit free lists
- Segregated free lists
- Garbage collection (brief overview)
- Memory-related perils and pitfalls
Garbage collection (brief overview)

- Allocated blocks that are no longer needed are called garbage.

- *Garbage collection:* automatic reclamation of heap-allocated storage—application never has to free

```c
void foo() {
    int *p = malloc(128);
    return; /* p block is now garbage */
}
```
Implicit Memory Management: Garbage Collection

- Common in functional languages, scripting languages, and modern object oriented languages:
  - Lisp, ML, Java, Perl, Mathematica
  - Easy to do if pointers are managed by the system (JVM/etc.)

- Variants ("conservative" garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage
    - Free usage of pointers makes it difficult
Garbage Collection

- **When to clean up memory?**
  - Future usage is unknown because it depends on conditionals
  - But we can tell that certain blocks cannot be used if there are no pointers to them

- **Must make certain assumptions about pointers**
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers
    (e.g., by coercing them to an `int`, and then back again)
Memory as a Graph

- **We view memory as a directed graph**
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called *root* nodes

![Root nodes](image1)

![Heap nodes](image2)

- root
- reachable
- Not-reachable (garbage)
Classical GC Algorithms (not discussed)

- Mark-and-sweep collection (McCarthy, 1960)
- Reference counting (Collins, 1960)
- Copying collection (Minsky, 1963)
- Generational Collectors (Lieberman and Hewitt, 1983)

For more information:
Mark and Sweep Collecting

- Can build on top of malloc/free package
  - Allocate using `malloc` until you “run out of space”
- When out of space:
  - Use extra `mark bit` in the head of each block
  - **Mark:** Start at roots and set mark bit on each reachable block
  - **Sweep:** Scan all blocks and free blocks that are not marked

![Memory Refs Diagram](image-url)

Note: arrows here denote memory refs, not free list ptrs.
Assumptions For a Simple Implementation

- **Application**
  - `new(n)`: returns pointer to new block with all locations cleared
  - `read(b,i)`: read location i of block b into register
  - `write(b,i,v)`: write v into location i of block b

- **Each block will have a header word**
  - addressed as `b[-1]`, for a block b
  - Used for different purposes in different collectors

- **Instructions used by the Garbage Collector**
  - `is_ptr(p)`: determines whether p is a pointer
  - `length(b)`: returns the length of block b, not including the header
  - `get_roots()`: returns all the roots
Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

```c
ptr mark(ptr p) {
    if (!is_ptr(p)) return; // do nothing if not pointer
    if (markBitSet(p)) return; // check if already marked
    setMarkBit(p); // set the mark bit
    for (i=0; i < length(p); i++) // call mark on all words
        mark(p[i]); // in the block
    return;
}
```

Sweep using lengths to find next block

```c
ptr sweep(ptr p, ptr end) {
    while (p < end) {
        if markBitSet(p)
            clearMarkBit();
        else if (allocateBitSet(p))
            free(p);
        p += length(p);
    }
```
Conservative Mark & Sweep in C

- A “conservative garbage collector” for C programs
  - `is_ptr()` determines if a word is a pointer by checking if it points to an allocated block of memory
  - But, in C pointers can point to the middle of a block

- So how to find the beginning of the block?
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - Balanced-tree pointers can be stored in header (use two additional words)
Topics

- Explicit free lists
- Segregated free lists
- Garbage collection
- C and Memory-related perils and pitfalls
C Pointer Declarations: Test Yourself!

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int *p</code></td>
<td><code>p</code> is a pointer to <code>int</code></td>
</tr>
<tr>
<td><code>int *(p[13])</code></td>
<td><code>p</code> is an array[13] of pointer to <code>int</code></td>
</tr>
<tr>
<td><code>int **p</code></td>
<td><code>p</code> is a pointer to a pointer to an <code>int</code></td>
</tr>
<tr>
<td><code>int (*p)[13]</code></td>
<td><code>p</code> is a pointer to an array[13] of <code>int</code></td>
</tr>
<tr>
<td><code>int *f()</code></td>
<td><code>f</code> is a function returning a pointer to <code>int</code></td>
</tr>
<tr>
<td><code>int (*f)()</code></td>
<td><code>f</code> is a pointer to a function returning <code>int</code></td>
</tr>
</tbody>
</table>

- `->`, `()`, and `[]` have high precedence, with `*` and `&` just below
- Unary `+`, `−`, and `*` have higher precedence than binary forms
### C Pointer Declarations: Test Yourself!

<table>
<thead>
<tr>
<th>Pointer Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int *p</code></td>
<td>p is a pointer to int</td>
</tr>
<tr>
<td><code>int *(p[13])</code></td>
<td>p is an array[13] of pointer to int</td>
</tr>
<tr>
<td><code>int **p</code></td>
<td>p is a pointer to a pointer to an int</td>
</tr>
<tr>
<td><code>int (*p)[13]</code></td>
<td>p is a pointer to an array[13] of int</td>
</tr>
<tr>
<td><code>int *f()</code></td>
<td>f is a function returning a pointer to int</td>
</tr>
<tr>
<td><code>int (*f)()</code></td>
<td>f is a pointer to a function returning int</td>
</tr>
<tr>
<td><code>int (**f())[13]()</code></td>
<td>f is a function returning ptr to an array[13] of pointers to functions returning int</td>
</tr>
<tr>
<td><code>int (**x[3])[5]()</code></td>
<td>x is an array[3] of pointers to functions returning pointers to array[5] of ints</td>
</tr>
</tbody>
</table>

- `->, ( ), and [ ]` have high precedence, with `*` and `&` just below
- Unary `+`, `−`, and `*` have higher precedence than binary forms

Source: K&R Sec 5.12
Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks
Dereferencing Bad Pointers

- The classic `scanf` bug

```c
int val;
...
scanf("%d", val);
```

- Problem: ?
- Fix: ?
Dereferencing Bad Pointers

- The classic `scanf` bug

```c
int val;
...
scanf("%d", val);
```

- Problem: `scanf` stores data at an address
- Fix: ?
Dereferencing Bad Pointers

- The classic `scanf` bug

```c
int val;

...  

scanf("%d", val);
```

- Problem: `scanf` stores data at an address
- Fix:

```c
int val;

...  

scanf("%d", &val);
```
Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```c
/* return y = Ax */
int *matvec(int **A, int *x) {
  int *y = malloc(N*sizeof(int));
  int i, j;

  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      y[i] += A[i][j]*x[j];
  return y;
}
```

- Fix: ?
Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```c
/* return y = Ax */
int *matvec(int **A, int *x) {
    int *y = calloc(N*sizeof(int));
    int i, j;

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            y[i] += A[i][j]*x[j];
    return y;
}
```

- Fix: zero y[i] or use calloc()
Overwriting Memory

- Allocating the (possibly) wrong sized object

```c
// creating an array of pointers to an array of ints:
int **p;
p = malloc(N*sizeof(int));
for (i=0; i<N; i++) {
    p[i] = malloc(M*sizeof(int));
}
```

- Problem: ?
Overwriting Memory

- Allocating the (possibly) wrong sized object

```c
// creating an array of pointers to an array of ints:
int **p;
p = malloc(N*sizeof(int));
for (i=0; i<N; i++) {
    p[i] = malloc(M*sizeof(int));
}
```

- Problem: ?
Overwriting Memory

- Allocating the (possibly) wrong sized object

```c
// creating an array of pointers to an array of ints:
int **p;

p = malloc(N*sizeof(int));
for (i=0; i<N; i++) {
    p[i] = malloc(M*sizeof(int));
}
```

- Problem: if sizeof(int) != sizeof(int*)
- Fix: ?
Overwriting Memory

- Allocating the (possibly) wrong sized object

// creating an array of pointers to an array of ints:
int **p;

p = malloc(N*sizeof(int));
for (i=0; i<N; i++) {
    p[i] = malloc(M*sizeof(int));
}

- Problem: if sizeof(int) != sizeof(int*)
- Fix:

... 
p = malloc(N*sizeof(int *));
...
Overwriting Memory

- Not checking the max string size

```c
char s[8];
int i;

gets(s);
```

- Problem: ?
Overwriting Memory

- Not checking the max string size

```c
char s[8];
int i;

gets(s); /* reads “123456789” from stdin */
```

- Problem: Basis for classic buffer overflow attacks
- Fix: ?
Overwriting Memory

- Not checking the max string size

```c
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

- Problem: Basis for classic buffer overflow attacks
- Fix: use fgets()
Overwriting Memory

- Misunderstanding pointer arithmetic

```c
int *search(int *p, int val) {
    while (*p && *p != val)
        p += sizeof(int);
    return p;
}
```

- Problem: ?
Overwriting Memory

- Misunderstanding pointer arithmetic

```c
int *search(int *p, int val) {
    while (*p && *p != val)
        p += sizeof(int);
    return p;
}
```

- Problem: ?
Overwriting Memory

- Misunderstanding pointer arithmetic

```c
int *search(int *p, int val) {
    while (*p && *p != val)
        p += sizeof(int);
    return p;
}
```

- Problem: advancing pointer by 4 integers
Overwriting Memory

- Misunderstanding pointer arithmetic

```c
int *search(int *p, int val) {
    while (*p && *p != val)
        p += sizeof(int);
    return p;
}
```

- Problem: advancing pointer by 4 integers
- Fix: p++;
Overwriting Memory

- Referencing a pointer instead of the object it points to

```c
// Removes first item from binary heap
int *binheapDelete(int **binheap, int *size) {
    int *packet;
    packet = binheap[0];
    // move last element to front
    binheap[0] = binheap[*size - 1];
    *size--; // decrement size by one
    heapify(binheap, *size, 0);
    return(packet);
}
```

- Problem: ?
Overwriting Memory

- Referencing a pointer instead of the object it points to

```c
// Removes first item from binary heap
int *binheapDelete(int **binheap, int *size) {
    int *packet;
    packet = binheap[0];
    // move last element to front
    binheap[0] = binheap[*size - 1];
    *size--; // decrement size by one
    heapify(binheap, *size, 0);
    return(packet);
}
```

- Problem: decrements pointer by one instead of the value being pointed to
- Fix: ?
Overwriting Memory

- Referencing a pointer instead of the object it points to

```c
// Removes first item from binary heap
int *binheapDelete(int **binheap, int *size) {
    int *packet;
    packet = binheap[0];
    // move last element to front
    binheap[0] = binheap[*size - 1];
    *size--; // decrement size by one
    heapify(binheap, *size, 0);
    return(packet);
}
```

- Problem: decrements pointer by one instead of the value being pointed to
- Fix: (*size)--;
Overwriting Memory

- Off-by-one error

```c
int **p;
p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
    p[i] = malloc(M*sizeof(int));
}
```

- Problem/Fix: ?
Overwriting Memory

- **Off-by-one error**

```c
int **p;
p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
    p[i] = malloc(M*sizeof(int));
}
```

- **Problem/Fix:** tries to store an item past p memory space

```c
int **p;
p = malloc(N*sizeof(int *));

for (i=0; i<N; i++) {
    p[i] = malloc(M*sizeof(int));
}
```
Referencing Nonexistent Variables

```c
int *foo () {
    int val;
    return &val;
}
```

Problem: ?
Referencing Nonexistent Variables

```c
int *foo () {
    int val;
    return &val;
}
```

- Problem: Forgetting that local variables disappear when a function returns
Freeing Blocks Multiple Times

Problem: Double free()

```c
x = malloc(N*sizeof(int));
   <manipulate x>
free(x);

y = malloc(M*sizeof(int));
   <manipulate y>
free(x);
```
Referencing Freed Blocks

\[
x = \text{malloc}(N*\text{sizeof(int))};
<\text{manipulate x}>
\text{free}(x);
...\
y = \text{malloc}(M*\text{sizeof(int))};
\text{for } (i=0; i<M; i++)
\quad y[i] = x[i]++;
\]

- May not show itself till later...
Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer!

```c
foo() {
    int *x = malloc(N*sizeof(int));
    ...
    return;
}
```
Failing to Free Blocks (Memory Leaks)

```c
struct list {
    int val;
    struct list *next;
};

foo() {
    struct list *head = malloc(sizeof(struct list));
    head->val = 0;
    head->next = NULL;
    <create and manipulate the rest of the list>
    ...
    free(head);
    return;
}
```

Problem: ?
Failing to Free Blocks (Memory Leaks)

```c
struct list {
    int val;
    struct list *next;
};

foo() {
    struct list *head = malloc(sizeof(struct list));
    head->val = 0;
    head->next = NULL;
    <create and manipulate the rest of the list>
    ...
    free(head);
    return;
}
```

Problem: All other nodes may be left in heap with no reference.
Dealing With Memory Bugs

- Compiler warnings
  - Get all Warnings and compile with debug symbols: `gcc -Wall -g`
  - Good for catching invalid pointer casting
Dealing With Memory Bugs

- **Compiler warnings**
  - Get all Warnings and compile with debug symbols: `gcc -Wall -g`
  - Good for catching invalid pointer casting

- **Conventional debugger (gdb)**
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
Dealing With Memory Bugs

- **Compiler warnings**
  - Get all Warnings and compile with debug symbols: `gcc -Wall -g`
  - Good for catching invalid pointer casting

- **Conventional debugger (gdb)**
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs

- **Some malloc implementations contain checking code**
  - glibc malloc: `export MALLOC_CHECK_=3`
    - Aborts on any heap corruption (find errors early)
Dealing With Memory Bugs

Binary translator: valgrind (Linux)
- Powerful debugging and analysis technique
- Rewrites text section of executable object file
- Can detect all errors as debugging malloc
- Can also check each individual reference at runtime
  - Bad pointers
  - Overwriting
  - Referencing outside of allocated block
Summary

- Dynamic memory allocator manages the heap
- Dynamic memory allocator is part of the user-space
- The allocator has two main goals:
  - reaching higher throughput (operations per second)
  - better memory utilization (i.e. reduces fragmentation).
- B&O Sec. 9.9.12 is very useful