Programming in C and C++

Lecture 7: Reference Counting and Garbage Collection

David J Greaves and Alan Mycroft
(Materials by Neel Krishnaswami)

The C API for Dynamic Memory Allocation

- In the previous lecture, we saw how to use arenas and ad-hoc graph traversals to manage memory when pointer graphs contain aliasing or cycles
- These are not the only idioms for memory management in C!
- Two more common patterns are *reference counting* and *type-specific garbage collectors*.

A Tree Data Type

```c
struct node {
  int value;
  struct node *left;
  struct node *right;
};
typedef struct node Tree;
```

- This is still the tree type from Lab 4.
- It has a value, a left subtree, and a right subtree
- An empty tree is a `NULL` pointer.

Construct Nodes of a Tree

```c
Tree *node(int value, Tree *left, Tree *right) {
  Tree *t = malloc(sizeof(tree));
  t->value = value;
  t->right = right;
  t->left = left;
  return t;
}
```

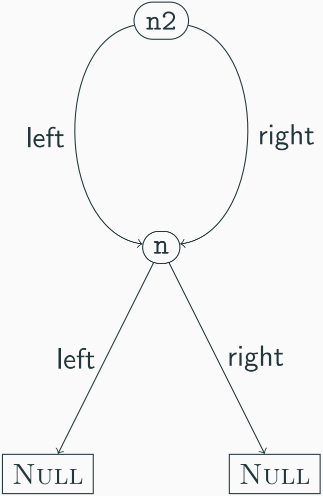
1. Allocate a pointer to a tree struct
2. Initialize the `value` field
3. Initialize the `right` field
4. Initialize the `left` field
5. Return the initialized pointer!

A Directed Acyclic Graph (DAG)

```
1     Tree *n = node(2, NULL, NULL);
2     Tree *n2 =
3       node(1, n, n); // n repeated!
```

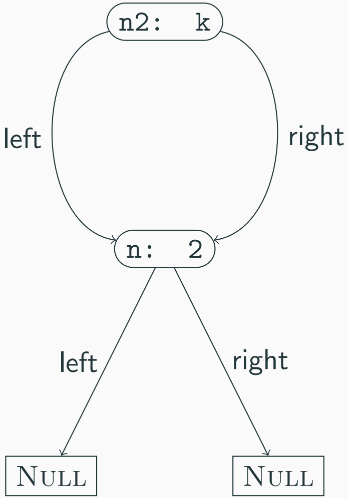
1. We allocate `n` on line 1
2. On line 2, we create `n2` whose `left` *and* `right` fields are `n`.
3. Hence `n2->left` and `n2->right` are said to *alias* – they are two pointers aimed at the same block of memory.

The shape of the graph



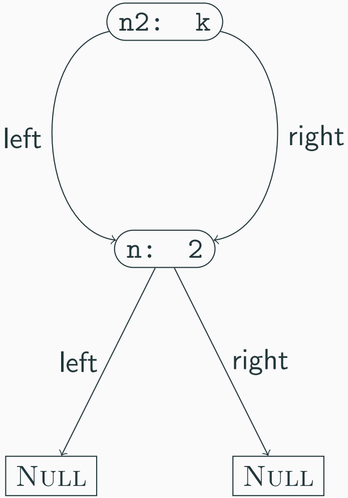
- `node1` has *two* pointers to `node2`
- This is a directed acyclic graph, not a tree.
- A recursive free of the tree `n2` will try to free `n` twice.

The Idea of Reference Counting



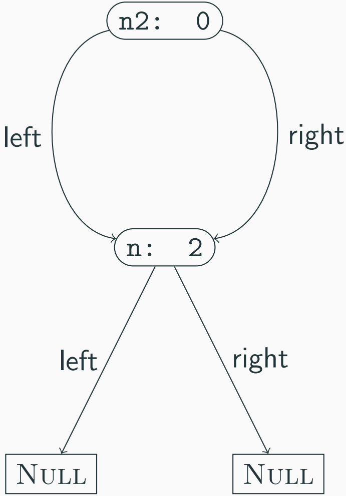
1. The problem: freeing things with two pointers to them twice
2. Solution: stop doing that
3. Keep track of the number of pointers to an object
4. Only free when the count reaches zero

How Reference Counting Works



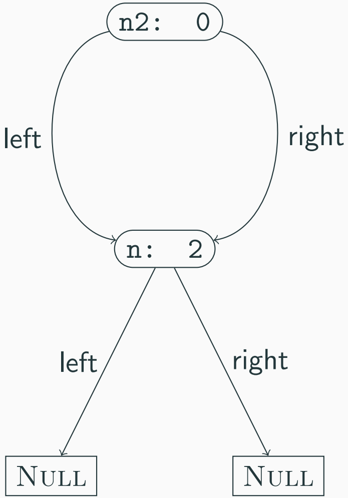
1. We start with k references to `n2`

How Reference Counting Works



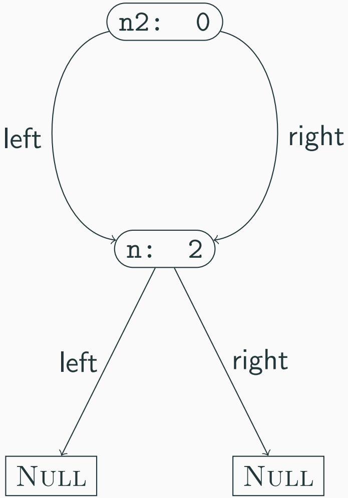
1. We start with k references to `n2`
2. Eventually k becomes 0

How Reference Counting Works



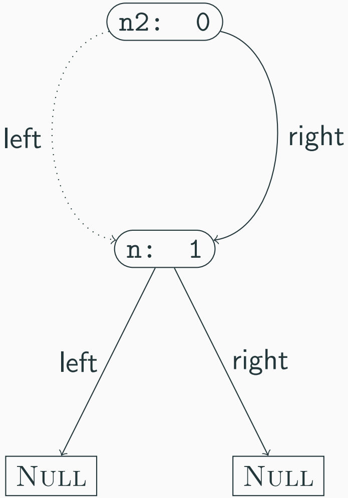
1. We start with k references to `n2`
2. Eventually k becomes 0
3. It's time to delete `n2`

How Reference Counting Works



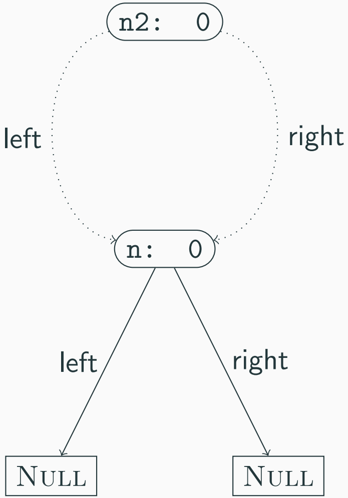
1. We start with k references to `n2`
2. Eventually k becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to

How Reference Counting Works



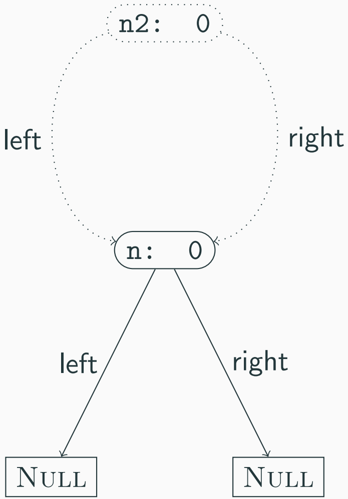
1. We start with k references to `n2`
2. Eventually k becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to

How Reference Counting Works



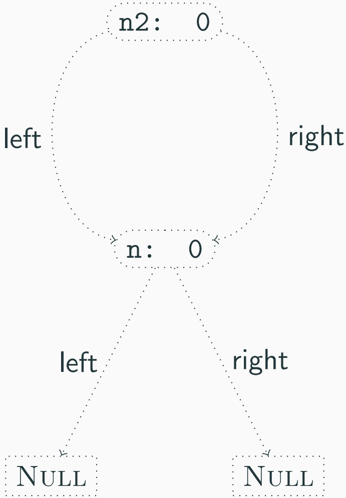
1. We start with k references to `n2`
2. Eventually k becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to
5. Then delete `n2`

How Reference Counting Works



1. We start with k references to `n2`
2. Eventually k becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to
5. Then delete `n2`

How Reference Counting Works



1. We start with k references to `n2`
2. Eventually k becomes 0
3. It's time to delete `n2`
4. Decrement the reference count of each thing `n2` points to
5. Then delete `n2`
6. Recursively delete `n`

The Reference Counting API

```
struct node {
  unsigned int rc;
  int value;
  struct node *left;
  struct node *right;
};
typedef struct node Node;

const Node *empty = NULL;
Node *node(int value,
           Node *left,
           Node *right);
void inc_ref(Node *node);
void dec_ref(Node *node);
```

- We add a field rc to keep track of the references.
- We keep the same node constructor interface.
- We add a procedure inc_ref to increment the reference count of a node.
- We add a procedure dec_ref to decrement the reference count of a node.

Reference Counting Implementation: `node()`

```
1  Node *node(int value,
2             Node *left,
3             Node *right) {
4    Node *r = malloc(sizeof(Node));
5    r->rc = 1;
6    r->value = value;
7
8    r->left = left;
9    inc_ref(left);
10
11   r->right = right;
12   inc_ref(right);
13   return r;
14 }
```

- On line 4, we initialize the `rc` field to 1. (Annoyingly, this is a rather delicate point!)
- On line 8-9, we set the `left` field, and increment the reference count of the pointed-to node.
- On line 11-12, we do the same to `right`

Reference Counting Implementation: `inc_ref()`

```
1 void inc_ref(Node *node) {
2   if (node != NULL) {
3     node->rc += 1;
4   }
5 }
```

- On line 3, we increment the `rc` field (if nonnull)
- That's it!

Reference Counting Implementation: `dec_ref()`

```c
1  void dec_ref(Node *node) {
2    if (node != NULL) {
3      if (node->rc > 1) {
4        node->rc -= 1;
5      } else {
6        dec_ref(node->left);
7        dec_ref(node->right);
8        free(node);
9      }
10   }
11 }
```

- When we decrement a reference count, we check to see if we are the last reference (line 3)
- If not, we just decrement the reference count (line 4)
- If so, then decrement the reference counts of the children (lines 6-7)
- Then free the current object. (line 8)

Example 1

```
1  Node *complete(int n) {
2    if (n == 0) {
3      return empty;
4    } else {
5      Node *sub = complete(n-1);
6      Node *result =
7        node(n, sub, sub);
8      dec_ref(sub);
9      return result;
10   }
11 }
```

- `complete(n)` builds a complete binary tree of depth n
- Sharing makes memory usage $O(n)$
- On line 5, makes a recursive call to build subtree.
- On line 6, builds the tree
- On line 8, call `dec_ref(sub)` to drop the stack reference `sub`
- On line 9, *don't* call `dec_ref(result)`

Example 1 – mistake 1

```c
Node *complete(int n) {
  if (n == 0) {
    return empty;
  } else {
    Node *sub = complete(n-1);
    Node *result =
      node(n, sub, sub);
    // dec_ref(sub);
    return result;
  }
}
```

- If we forget to call `dec_ref(sub)`, we get a memory leak!
- `sub` begins with a refcount of 1
- `node(sub, sub)` bumps it to 3
- If we call `dec_ref(complete(n))`, the outer node will get freed
- But the children will end up with an `rc` field of 1

Example 1 – mistake 2

```
1  Node *complete(int n) {
2    if (n == 0) {
3      return empty;
4    } else {
5      return node(n,
6                  complete(n-1),
7                  complete(n-1));
8    }
9  }
```

- This still leaks memory!
- `complete(n-1)` begins with a refcount of 1
- The expression on lines 5-7 bumps each subtree to a refcount of 2
- If we call `free(complete(n))`, the outer node will get freed
- But the children will end up with an `rc` field of 1

Design Issues with Reference Counting APIs

- The key problem: *who is responsible for managing reference counts?*
- Two main options: sharing references vs transferring references
- Both choices work, but must be made consistently
- To make this work, API must be documented very carefully
    - Good example: Python C API
    - `https://docs.python.org/3/c-api/intro.html#objects-types-and-reference-counts`

Mitigations: Careful Use of Getters and Setters

```c
Node *get_left(Node *node) {
  inc_ref(node->left);
  return(node->left);
}

void set_left(Node *node,
              Node *newval) {
  inc_ref(newval);
  dec_ref(node->left);
  node->left = newval;
}
```

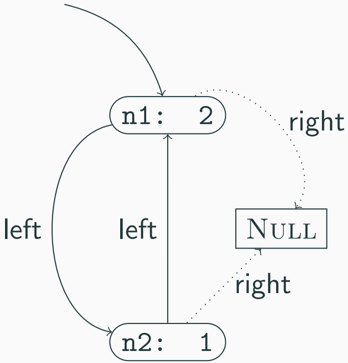
- The `get_left()` function returns the left subtree, but also increments the reference count
- The `set_left()` function updates the left subtree, incrementing the reference count to the new value and decrementing the reference

Cycles: A Fundamental Limitation on Reference Counting

```
Node *foo() {
  Node *n1 = node(1, NULL, NULL);
  Node *n2 = node(2, NULL, NULL);
  set_left(node1, node2);
  set_left(node2, node1);
  dec_ref(n2);
  return node1;
}
```

What does a call to `foo()` build?

A Cyclic Object Graph



- `n1->rc` is 2, since `n2` points to it
- `n2->rc` is 1, since `n1` points to it
- This is a cyclic graph
- Even though there is only 1 external reference to `n1`, `n1->rc` is 2.
- Hence `dec_ref(foo())` will not free memory!
- Reference counting *cannot collect cycles*

Garbage Collection: Dealing with Cycles

- In ML or Java, we don't have to worry about cycles or managing reference counts explicitly
- We rely on a *garbage collector* to manage memory automatically
- In C, we can *implement* garbage collection to manage memory

GC API – Data structures

```c
struct node {
  int value;
  struct node *left;
  struct node *right;
  bool mark;
  struct node *next;
};
typedef struct node Node;

struct root {
  Node *start;
  struct root *next;
};
typedef struct root Root;

struct alloc {
  Node *nodes;
  Root *roots;
};
typedef struct alloc Alloc;
```

- `Node *` are node objects, but augmented with a `mark` bit (Lab 5) and a `next` link connecting all allocated nodes
- A `Root *` is a node we don't want to garbage collect. Roots are also in a linked list
- An allocator `Alloc *` holds the head of the lists of nodes and roots

GC API – Procedures

```c
Alloc *make_allocator(void);
Node *node(int value,
           Node *left,
           Node *right,
           Alloc *a);
Root *root(Node *node, Alloc *a);
void gc(Alloc *a);
```

- `make_allocator` creates a fresh allocator
- `node(n, l, r, a)` creates a fresh node in allocator a (as in the arena API)
- `root(n)` creates a new root object rooting the node `n`
- `gc(a)` frees all nodes unreachable from the roots

Creating a Fresh Allocator

```
Alloc *make_allocator(void) {
  Alloc *a = malloc(sizeof(Alloc));
  a->roots = NULL;
  a->nodes = NULL;
  return a;
}
```

- Creates a fresh allocator with empty set of roots and nodes
- Invariant: no root or node is part of two allocators!
- (Could use global variables, but thread-unfriendly)

```
1  Node *node(int value,
2             Node *left,
3             Node *right,
4             Alloc *a) {
5    Node *r = malloc(sizeof(Node));
6    r->value = value;
7    r->left = left;
8    r->right = right;
9    //
10   r->mark = false;
11   r->next = a->nodes;
12   a->nodes = r;
13   return r;
14 }
```

- Lines 5-9 perform familiar operations: allocate memory (line 5) and initialize data fields (6-8)
- Line 10 initializes `mark` to `false`
- Lines 11-12 add new node to `a->nodes`

Creating a Root

```
1 Root *root(Node *node,
2            Alloc *a) {
3   Root *g =
4     malloc(sizeof(Root));
5   g->start = node;
6   g->next = a->roots;
7   a->roots = g;
8   return g;
9 }
```

- On line 4, allocate a new `Root` struct `g`
- On line 5, set the `start` field to the `node` argument
- On lines 6-7, attach `g` to the `roots` of the allocator `a`
- Now the allocator knows to treat the root as always reachable

Implementing a Mark-and-Sweep GC

- Idea: split GC into two phases, *mark* and *sweep*
- In mark phase:
    - From each root, mark the nodes reachable from that root
    - I.e., set the `mark` field to true
    - So every reachable node will have a true mark bit, and every unreachable one will be set to false
- In sweep phase:
    - Iterate over every allocated node
    - If the node is unmarked, free it
    - If the node is marked, reset the mark bit to false

Marking

```c
void mark_node(Node *node) {
  if (node != NULL && !node->mark) {
    node->mark = true;
    mark_node(node->left);
    mark_node(node->right);
  }
}

void mark(Alloc *a) {
  Root *g = a->roots;
  while (g != NULL) {
    mark_node(g->start);
    g = g->next;
  }
}
```

- `mark_node()` function marks a node if unmarked, and then recursively marks subnodes
- Just like in lab 6!
- `mark()` procedure iterates over the roots, marking the nodes reachable from it.
- If a node is not reachable from the `a->roots` pointer, it will stay `false`

Sweeping

```
1  void sweep(Alloc *a) {
2    Node *n = a->nodes;
3    Node *live = NULL;
4    while (n != NULL) {
5      Node *tl = n->next;
6      if (!(n->mark)) {
7        free(n);
8      } else {
9        n->mark = false;
10       n->next = live;
11       live = n;
12     }
13     n = tl;
14   }
15   a->nodes = live;
16 }
```

- On line 2, get a pointer to *all allocated nodes* via `a->nodes`
- On line 3, create a new empty list of `live` nodes
- On lines 4-14, iterate over each allocated node
- On line 6, check to see if the node is unmarked
- If unmarked, free it (line 8)
- If marked, reset the mark bit and add it to the `live` list (9-11)
- On line 15, update `a->nodes` to the still-live `live` nodes

The `gc()` routine

```
void gc(Alloc *a) {
  mark(a);
  sweep(a);
}
```

- `gc(a)` just marks and sweeps!
- To use the gc, we allocate nodes as normal
- Periodically, invoke `gc(a)` to clear out unused nodes
- That's it!

Design Considerations

- This kind of custom GC is quite slow relative to ML/Java gcs
- However, simple and easy to implement (only 50 lines of code!)
- No worries about cycles or managing reference counts
- Worth considering using the Boehm gc if gc in C/C++ is needed:
    - `https://www.hboehm.info/gc/`
    - Drop-in replacement for malloc!
- Still useful when dealing with interop between gc'd and manually-managed languages (eg, DOM nodes in web browsers)