CSE 333 Section 7 - Memory Diagram Review, Smart Pointers

Welcome back to section! We're glad that you're here :)

Memory Diagram Review

Memory diagrams, sometimes called box-and-arrow diagrams, visually describe the state of the program.

A good memory diagram will have:

- An area for the stack and (if dynamic memory) heap, ideally both labeled
- Be sure to label stackframes!
- Variables will be located in their stack frames or the heap. Every variable will have its own box, labeled with its name. Its value is drawn within the box.
- If a struct/class's fields are clear from context, there's no need to label them; if it's unclear, then label

<u>Exercise 1</u>

Recall the linked list from ex9.5. For convenience, we've repeated the relevant parts of `ll.h` and `main.c` here. Additionally, we've given a sample (non-buggy!) implementation of `ll.c`.

```
// ll.h

typedef struct llnode {
  struct llnode *next;
  int payload;
} LinkedListNode;

typedef struct ll {
  LinkedListNode *head;
  int size;
} LinkedList;

typedef void(*PayloadFn)(int *payload);

LinkedList *LinkedList_Allocate();
void LinkedList_Push(LinkedList *l, int i);
void LinkedList_Iterate(LinkedList *l, PayloadFn fn);
void LinkedList_Deallocate(LinkedList *l);
```

```
// main.c

void print_payload(int *i) {
  fprintf(stderr, "%d ", *i);
}

int main(void) {
  int i;
  LinkedList *ll = LinkedList_Allocate();

  printf("Please enter a list of integers you'd like reversed: ");
  while (1) {
    if (scanf("%d", &i) != 1) break;
```

```
    LinkedList_Push(ll, i);
  }

  printf("\nYour reversed numbers are:\n");
  LinkedList_Iterate(ll, &print_payload);
  printf("\n");

  LinkedList_Deallocate(ll);
  return 0;
}
```

```
// ll.c

LinkedList *LinkedList_Allocate() {
  LinkedList *l = (LinkedList *)malloc(sizeof(LinkedList));
  l->head = NULL;
  l->size = 0;
  return l;
}

void LinkedList_Push(LinkedList *l, int i) {
  LinkedListNode *n = (LinkedListNode*)malloc(sizeof(LinkedListNode));
  n->next = l->head;
  n->payload = i;

  l->head = n;
  l->size++;
}

void LinkedList_Iterate(LinkedList *l, PayloadFn fn) {
  LinkedListNode *n = l->head;
  while (n != NULL) {
    fn( &(n->payload) );
    n = n->next;
  }
}

void LinkedList_Deallocate(LinkedList *l) {
  LinkedListNode *n = l->head;
  while (n != NULL) {
    LinkedListNode *nxt = n->next;
    free(n);
    n = nxt;
  }
}
```

(continued on next page)

Assume that you've set a breakpoint at the **first** call to print_payload(). Draw the memory diagram at that breakpoint, assuming that the program's input was "10 20 30".

Smart Pointers!

`std::unique_ptr` – Uniquely manages a raw pointer by disabling `cctor` and `op=`

- Used when you want to declare unique ownership of a pointer

`std::shared_ptr` – Uses reference counting to determine when to delete a managed raw pointer

- Use when multiple pointers need to “own” the heap resource *simultaneously*

`std::weak_ptr` – Used in conjunction with `shared_ptr` but does **not** contribute to reference count

<u>Exercise 2</u>

Consider the `IntNode` struct below. Convert the `IntNode` struct to be “smart”.
Should each field be a `unique_ptr`, `shared_ptr`, or `weak_ptr`? Why?

```
#include <memory>
using std::shared_ptr;
using std::unique_ptr;
using std::weak_ptr;

struct IntNode {
  IntNode(int* val, IntNode* node): value(val), next(node) {}

  ~IntNode() { delete value; }

  int* value;
  IntNode* next;
};
```

After the conversion, draw a memory diagram with the reference count for blocks of memory.

```
#include <iostream>

using std::cout;
using std::endl;
using std::shared_ptr;

int main() {
  shared_ptr<IntNode> head =
      shared_ptr<IntNode>(new IntNode(new int(351), nullptr));
  head->next = shared_ptr<IntNode>(new IntNode(new int(333),
                                                    nullptr));
  shared_ptr<IntNode> iter = head;
  while (iter != nullptr) {
    cout << *(iter->value) << endl;
    iter = iter->next;
  }
}
```