Section 6. Memory Management, Pointers, and Linked Lists

Section materials curated by Neel Kishnani, drawing upon materials from previous quarters.

This week's section exercises delve deep into the details of pointers and memory management in C++. This is an opportunity to get into the nitty gritty of things and to get closer towards gaining ultimate power over your computer! You will also gain valuable practice with linked lists, which are a new way of storing and organizing data that takes advantage of the power of pointers. Linked lists are definitely a tricky subject, but if you draw lots of diagrams and really nail down your pointer fundamentals, you'll be on the road to success. The topics covered in section this week will show up on assignment 6.

Remember that every week we will also be releasing a Qt Creator project containing starter code and testing infrastructure for that week's section problems. When a problem name is followed by the name of a **.cpp** file, that means you can practice writing the code for that problem in the named file of the Qt Creator project. Here is the zip of the section starter code:

Starter code

For all the problems in this handout, assume the following structures have been declared:

```
struct Node {
    int data;
    Node *next;
};

struct StringNode {
    string data;
    StringNode *next;
};

struct DoubleNode {
    double data;
    DoubleNode *next;
};
```

You can also assume the following utility functions have been defined as well:

```
/* Prints the contents of a linked list, in order. */
void printList(Node* list) {
    for (Node* cur = list; cur != nullptr; cur = cur->next) {
        cout << cur->data << endl;
    }
}

/* Frees all the memory used by a linked list. */
void deleteList(Node* list) {
    while (list != nullptr) {
        /* Store where to go next, since we're about to blow up our linked
         * list node.
         */
        Node *next = list->next;
        delete list;
        list = next;
    }
}
```

1) Some Pointers on Cats

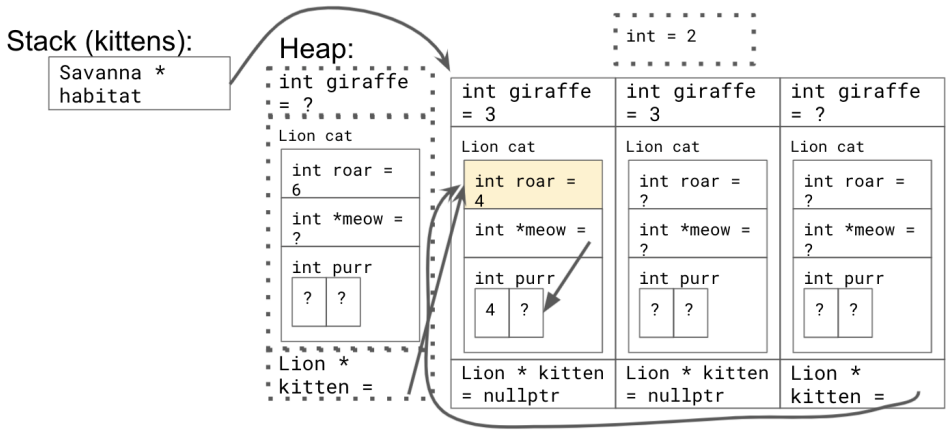
Topics: Pointer tracing and memory diagrams

Trace through the following function and draw the program's memory at the designated spot. Indicate which variables are on the stack and which are on the heap, and indicate orphaned memory. Indicate with a question mark (?) memory that we don't know the values of.

```
struct Lion {
        int roar;
        int *meow;
        int purr[2];
};

struct Savanna {
        int giraffe;
        Lion cat;
        Lion *kitten;
};

Lion *explore(Savanna *prairie) {
        Lion *leader = &(prairie->cat);
        leader->meow = new int;
        *(leader->meow) = 2;
        prairie = new Savanna;
        prairie->cat.roar = 6;
        prairie->kitten = leader;
        prairie->kitten->roar = 8;
        prairie->kitten->meow = &(prairie->kitten->purr[1]);
        leader->purr[0] = 4;
        return leader;
}

void kittens() {
        Savanna *habitat = new Savanna[3];
        habitat[1].giraffe = 3;
        habitat[1].kitten = nullptr;
        habitat[0] = habitat[1];
        habitat[2].kitten = explore(habitat);
        habitat[2].kitten->roar = 4;

        // DRAW THE MEMORY AS IT LOOKS HERE
}
```

Solution

Orphaned memory (memory on the heap that we no longer have access to but that was not freed) is represented with dotted lines. For a full walkthrough of the solution, check out: https://tinyurl.com/CatsPointers.



2) The Notorious RBQ, Revisited (`RingBufferQueue.h/.cpp`)

Topics: Classes, dynamic memory allocation, pointers

Remember our good friend the **RingBufferQueue** from last section? Check out the problem definition from last week if you want a quick refresher. Last time we visited the RBQ it had fixed capacity – that is, it couldn't grow in size after it was initially created. How limiting! With our newfound pointer and dynamic allocation skills, we can remove this limitation on the RBQ and make it fully functional!

Add functionality to the class **RingBufferQueue** from the previous section problem so that the queue resizes to an array twice as large when it runs out of space. In other words, if asked to enqueue when the queue is full, it will enlarge the array to give it enough capacity.

For example, say our queue can hold 5 elements and we enqueue the five values 10, 20, 30, 40, and 50. Our queue would look like this:

```
index    0    1    2    3    4
      +----+----+----+----+----+
value | 10 | 20 | 30 | 40 | 50 |
      +----+----+----+----+----+
         ^                   ^
         |                   |
       head                tail
```

If the client tries to enqueue a sixth element of 60, your improved queue class should grow to an array twice as large:

```
index    0    1    2    3    4    5    6    7    8    9
      +----+----+----+----+----+----+----+----+----+----+
value | 10 | 20 | 30 | 40 | 50 | 60 |    |    |    |    |
      +----+----+----+----+----+----+----+----+----+----+
         ^                        ^
         |                        |
       head                     tail
```

The preceding is the simpler case to handle. But what about if the queue has wrapped around via a series of enqueues and dequeues? For example, if the queue stores the following five elements:

```
index    0    1    2    3    4
      +----+----+----+----+----+
value | 40 | 50 | 10 | 20 | 30 |
      +----+----+----+----+----+
              ^    ^
              |    |
            tail head
```

If the client tries to add a sixth element of 60, you cannot simply copy the array as it is shown. If you do so, the head and wrapping will be broken. Instead, copy into the new array so that index 0 always becomes the new head of the queue. The picture will look the same as the previous one with the value 60 at index 5.

Write up the implementation of the new and improved **RingBufferQueue**. It should have the same members as in the previous problem, but with the new resizing behavior added. You may add new member functions to your class, but you should make them private.

Solution

RingBufferQueue.h

```
#pragma once

#include <iostream>
using namespace std;
class RingBufferQueue {
public:
    RingBufferQueue();
    ~RingBufferQueue();
    bool isEmpty() const;
    bool isFull() const;
    int size() const;
    void enqueue(int elem);
    int dequeue();
    int peek() const;
private:
    int* _elements;
    int _capacity;
    int _size;
    int _head;
    friend ostream& operator <<(ostream& out, const RingBufferQueue&
queue);
    void enlarge();
};
```

RingBufferQueue.cpp

```
#include "RingBufferQueue.h"

static int kDefaultCapacity = 5;
RingBufferQueue::RingBufferQueue() {
    _capacity = kDefaultCapacity;
    _elements = new int[_capacity];
    _head = 0;
    _size = 0;
}
RingBufferQueue::~RingBufferQueue() {
    delete[] _elements;
}
void RingBufferQueue::enqueue(int elem) {
    if (isFull()) {
        enlarge();
    }
    int tail = (_head + _size) % _capacity;
    _elements[tail] = elem;
    _size++;
}
int RingBufferQueue::dequeue() {
    int front = peek();
    _head = (_head + 1) % _capacity;
    _size--;
    return front;
}
int RingBufferQueue::peek() const {
    if (isEmpty()) {
        error("Can't peek at an empty queue!");
    }
    return _elements[_head];
}
bool RingBufferQueue::isEmpty() const {
    return _size == 0;
}
bool RingBufferQueue::isFull() const {
    return _size == _capacity;
}
int RingBufferQueue::size() const {
    return _size;
}
void RingBufferQueue::enlarge() {
    int *larger = new int[_capacity * 2];
    for (int i = 0; i < _size; i++) {
        larger[i] = _elements[(_head + i) % _capacity];
    }
    delete[] _elements;
    _elements = larger;
    _capacity *= 2;
    _head = 0;
}
ostream& operator <<(ostream& out, const RingBufferQueue& queue) {
    out << "{";
    if (!queue.isEmpty()) {
        // We can access the inner '_elements' member variables because
        // this operator is declared as a friend of the RingBufferQueue
class
        out << queue._elements[queue._head];
        for (int i = 1; i < queue.size(); i++) {
            int index = (queue._head + i) % queue._capacity;
            out << ", " << queue._elements[index];
        }
```

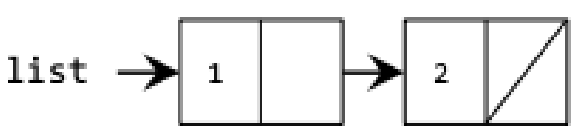
```
    }
    out << "}";
    return out;
}
```

Note: the following problems cover linked lists, which we'll be going over in lecture starting on Wednesday, May 11th. If you're looking for more practice on linked lists, these problems are a great resource!

3) What's the Code Do?

For each of the following diagrams, draw a picture of what the given nodes would look like after the given line of code executes. Does any memory get orphaned as a result of the operations? Assume the **Node** struct is the same as struct covered in lecture that stores integers.

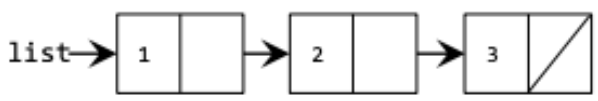
Diagram 1



Code Snippet 1

```
list->next = new Node;
list->next->data = 3;
```

Diagram 2

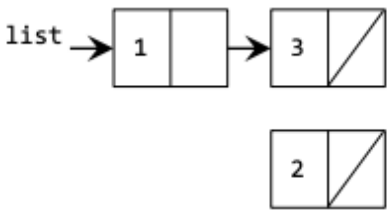


Code Snippet 2

```
list->next->next = nullptr;
```

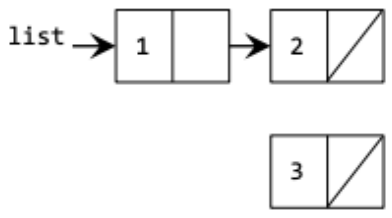
Solution

Example 1



This code produces one orphaned node with data 2.

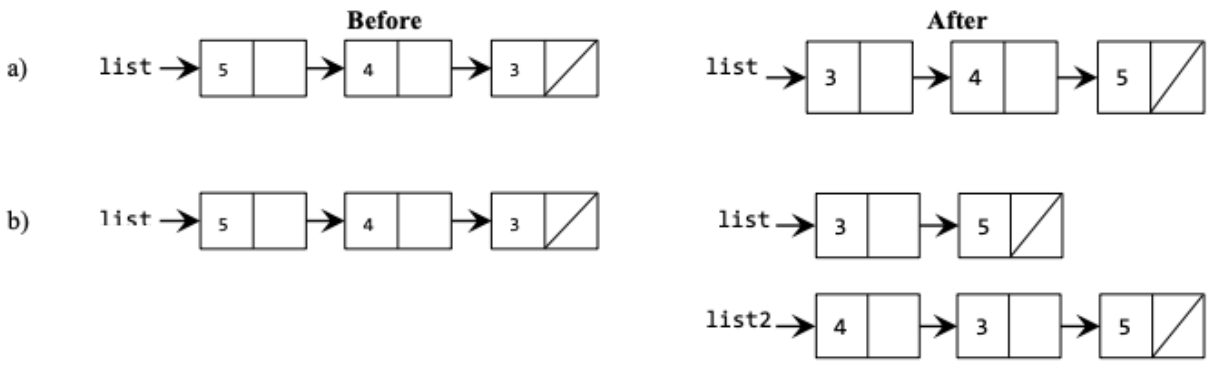
Example 2



This code produces one orphaned node with data 3.

4) Rewiring Linked Lists

For each of the following diagrams, write the code that will produce the given "after" result from the given "before" starting point by modifying the links between the nodes shown and/or creating new nodes as needed. There may be more than one way to write the code, but do **not** change the data field of any existing node. If a variable doesn't appear in the "after" picture, it doesn't matter what value it has after changes are made.



Solution

a)

```
Node *temp = list->next->next;
temp->next = list->next;
list->next->next = list;
list->next->next->next = nullptr;
list = temp;
```

b)

```
list->next->next->next = list;
list = list->next->next;
Node *list2 = list->next->next;
list->next->next = nullptr;
```

5) Linked List Mechanics (**warmup.cpp**)

This section handout is almost exclusively about linked lists, so before we jump into some of their applications, let's start off by reviewing some of the basic mechanics about how they work!

To begin with, let's imagine we have a linked list of integers. Write a function

```
int sumOfElementsIn(Node* list);
```

that adds up the values of all the elements in the linked list. Write this function two ways – first, do it iteratively; then, do it recursively. Which one did you think was easier to write? Why?

Next, write a function

```
Node* lastElementOf(Node* list);
```

that returns a pointer to the last element of a linked list (and reports an error if the list is empty). Again, write this function two ways, iteratively and recursively. Which one did you think was easier to write?

Solution

Summing List Elements

```
/* Iterative version */
int sumOfElementsIn(Node* list) {
    int result = 0;
    for (Node* curr = list; curr != nullptr; curr = curr->next) {
        result += curr->data;
    }
    return result;
}

/* Recursive version. */
int sumOfElementsIn(Node* list) {
    /* The sum of the elements in an empty list is zero. */
    if (list == nullptr) return 0;

    /* The sum of the elements in a nonempty list is the sum of the
elements in
    * the first node plus the sum of the remaining elements.
    */
    return list->data + sumOfElementsIn(list->next);
}
```

Finding the Last List Element

```
/* Iterative version */
Node* lastElementOf(Node* list) {
    if (list == nullptr) error("Empty lists have no last element.");

    /* Loop forward until the current node next pointer is null. That's
the
    * point where the list ends.
    */

    Node* result = list;
    while (result->next != nullptr) {
        result = result->next;
    }
    return result;
}

/* Recursive version. */
Node* lastElementOf(Node* list) {
    /* Base Case 1: The empty list has no last element. */
    if (list == nullptr) error("Nothing can come from nothing.");

    /* Base Case 2: The only element of a one-element list is the last
element. */

    if (list->next == nullptr) return list;

    /* Recursive Case: There's at least two nodes in this list. The last
element
    * of the overall list is the last element of the list you get when
you drop
    * off the first element.
    */
    return lastElementOf(list->next);
}
```


6) All out of Sorts (`sorted.cpp`)

Write a function that takes in a pointer to the front of a linked list of integers and returns whether or not the list that's pointed to is in sorted (nondecreasing) order. An empty list is considered to be sorted. You should implement your code to match the following prototype

```
bool isSorted(Node* front)
```

Solution

```
bool isSorted(Node* front) {
    if (front != nullptr) {
        Node* current = front;
        while (current->next != nullptr) {
            if (current->data > current->next->data) {
                return false;
            }
            current = current->next;
        }
    }
    return true;
}
```