CS61, Fall 2012 Section 2 Notes

SOLUTIONS (Week of 9/24-9/28)

- 0. Get source code for section [optional]
- 1: Variable Duration
- 2: Memory Errors
  - Common Errors with memory and pointers
    - Valgrind + GDB
  - Common Memory Errors When Using Malloc
  - How can you detect these errors? Your Problem Set!
- 3: Doubly Linked Lists
  - Insertion into a Doubly Linked List
  - Deletion from a Doubly Linked List
  - Checking if a node is in a Doubly Linked List
- 4: Garbage Collection
  - Garbage Collection Techniques
    - Mark and Sweep
    - Generation Garbage collection
    - Reference counting
- 5. Tips for Homework
  - 0. Pointer Arithmetic.
  - 1. Use structs to avoid complicated pointer arithmetic.
  - 2. Avoid pointer arithmetic on (void *) types!
  - 3. Use Git. Commit often and commit everything.

0. Get source code for section [optional]

To get the source code for all cs61 sections:

```
git clone git@code.seas.harvard.edu:cs61/cs61-section.git
```

This section's notes is in the s2/ directory.

1: Variable Duration

- static duration
  - variables will last the lifetime of the program (global variables)
- automatic duration
  - C will automatically handle the memory for these variables
  - variables will be placed on stack; allocated during a function call and freed when exiting
- dynamic duration
  - location for these variables are decided at runtime
  - placed in heap

- in C: must manage explicitly with malloc() and free()

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

//Prototypes
int doWork(void);

//Global Variables
int x = 3;

int doWork(void) {
  static int y; //static local variable
  if (x == 3) {
    y = 5;
  } else {
    y += 1;
  }
  return y;
}


int main(int argc, char *argv[]) {
  //static variable example
  for (int m = 0; m < 4; m++) {
    printf("y is: %d\n", doWork());
    x++;
  }

  x = 3;
  printf("y is once again: %d\n", doWork());

  //pointer example
  int *ptr = malloc(sizeof(int));
  *ptr = 15;
  printf("address of ptr itself: %p\n", &ptr);
  printf("address ptr points to: %p\n", ptr);
  printf("value of at that address: %d\n", *ptr);

  return 0;
}
```

Output:

```
y is: 5
y is: 6
y is: 7
y is: 8
y is once again: 5
```

```
address of ptr itself: 0x7fffc270b380 //(64 bit machine!)
address ptr points to: 0x601010
value of at that address: 15
```

Question: Where do these variables live? What are their durations?

- x
  - static duration
  - data section, not heap or stack
- y
  - static duration
  - data section, not heap or stack
- i
  - automatic duration
  - stack
- ptr
  - automatic duration
  - stack
- *ptr
  - dynamic duration
  - ptr lives in stack, *ptr lives in heap. ptr points to address in the heap!

2: Memory Errors

Common Errors with memory and pointers

Question: What is wrong with this code?
The code returns a pointer to the stack!! The stack frame containing the memory of x was reclaimed when get_int() returned, and then the space was reused by do_work()

```
int *get_int() {
  int x = 5;
  int *y = &x;
  return y;
}

int do_work(int a, int b) {
  int c = a + b;
  return c * c;
}

int main(void) {
  int *y = get_int();
  (void) do_work(10, 6);
  printf("%d\n", *y);
}
```

Output: 16

```
int main(void) {
  int *y = get_int();
  printf("%d\n", *y);
}
```

Output: 5
YOU CANNOT RETURN POINTERS TO THE STACK.

Question: What is wrong with this code? name is uninitialized! this will probably segfault.

```
int main(void) {
  char *name;
  strcpy(name, "Jeremy Lin");
  printf("%s\n", name);
}
```

Valgrind + GDB

Question: How can you debug the error in that code?
Valgrind + GDB!

```
valgrind --tool=memcheck -v ./uninitialized

Output:
…
==15537== Use of uninitialised value of size 8
==15537==    at 0x100011104: memcpy (mc_replace_strmem.c:893)
==15537==    by 0x100000EE1: __inline_strcpy_chk (in ./uninitialized)
==15537==    by 0x100000EAD: main (in ./uninitialized)
…
```

Valgrind tells you that you have an uninitialized value!!

Also, use GDB to step through your code and find out exactly where your code is breaking:

```
gdb -tui program // -tui option displays code as well, won't work on binary bomb!
```

GDB commands:

```
break #
next
step
disp <var>
undisp <var>
print <var>
```

Step through variables example to see the contents of x, y from first example

Question: What is wrong with this code?
misunderstanding pointer arithmetic: should be ptr += 1;

```
/** Assume the last element pointed to by ptr is 0. */
int *search(int *ptr, int val) {
  while (*ptr && *ptr != val) {
    ptr += sizeof(int);
  }
  return ptr;
}

int main(void) {
  int array[] = {5, 6, 7, 8, 9, 10, 11, 0};
  int *location = search(array, 7);
  printf("7 == %d?\n", *location);
}
```

Output: 7 == 0?

Common Memory Errors When Using Malloc

1. **Invalid free** - attempting to free a pointer that was not returned by malloc
    a. Examples:
        i. Freeing a pointer on the stack
        ii. Freeing a pointer to global data
        iii. Freeing a pointer randomly in the heap
    b. See test 11
2. **Double free** - attempting to free a pointer that was returned by malloc after it was already freed
    a. This is bad! The memory could have already been reclaimed and allocated to someone else.
    b. See test 13
3. **Boundary Overwrite** - writing past the requested block of memory
    a. See test 20

```
int *ptr = (int *) malloc(4 * sizeof(int));
for (int i = 0; i <= 4; i++) { // should be i < 4
  ptr[i] = i;
}
```

4. **Memory Leak** - not freeing all the pointers that were returned by malloc
    a. OK for short lived programs: the OS will reclaim the memory when the program terminates
    b. TERRIBLE for long running programs (e.g. your operating system, servers, your phone)
    c. See test 23

How can you detect these errors? Your Problem Set!

1. Detecting Invalid Frees
    a. Strategy: keep track of the current range of addresses of the heap

2. Double Free
    a. Strategy: Magic number in metadata to indicate whether something was allocated/freed
3. Boundary Overwrite
    a. Strategy: Have a footer and keep track if the number that was there was changed
4. Memory Leak
    a. Strategy: Maintain a list of currently allocated blocks--doubly linked lists!

3: Doubly Linked Lists

In order to find all memory leaks of a program, we need some way of knowing where all the allocations are. In order to make malloc and free O(1) operations (when they are successful), what kind of data structure should we use?

Question: What data structure offers creating nodes in constant time, and given pointers to the nodes, allows us to remove the nodes in constant time? Doubly Linked List!

Doubly linked lists are composed of nodes that contain pointers to both the previous and next node. Here is the struct for nodes:

```
typedef struct node {
  struct node *prev;
  int value;
  struct node *next;
} node;
```

For the nodes at the very front and very end, we set prev and next to point to NULL. To get a handle to the linked list, all we need is a pointer to the head (for doubly linked lists, a pointer to any node in the list will do!).

Question: What is the size of this struct on a 32 bit machine? 12 bytes

Insertion into a Doubly Linked List

We would like our insertion to be a constant time operation--thus it would be best if we always insert at the beginning or at the end of the list. For this exercise, we always insert new nodes into the beginning of the list. We maintain a global node pointer to the head of our list.

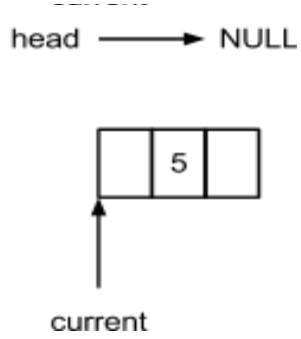
Question: When will be adding nodes to a doubly linked list in our pset? When we malloc nodes--we keep track of an allocated list.

Question: What are the different cases we have for insertion into a doubly linked list at the head?

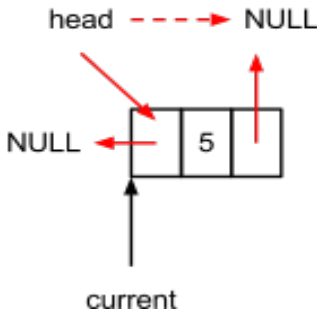
```
// Global variable that represents a pointer to the beginning of the list:
node *head = NULL;

/** Insert @a current at the head of the list pointed to by head. */
void insert_node(node *current) {
```

```
        if (head == NULL) {
            // case 1: list is empty
            current->prev = NULL;
            current->next = NULL;
            head = current;
        } else {
            // case 2: list is not empty
            current->prev = NULL;
            current->next = head;
            current->next->prev = current;
            head = current;
        }
}
```

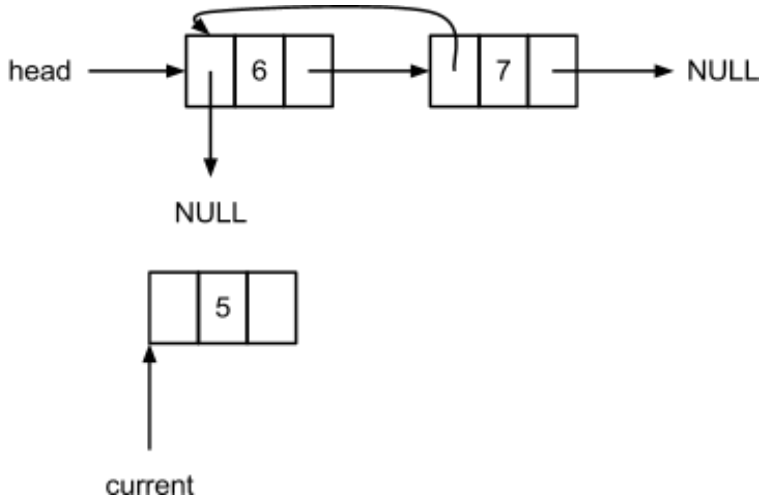
Case 1: The list is empty.



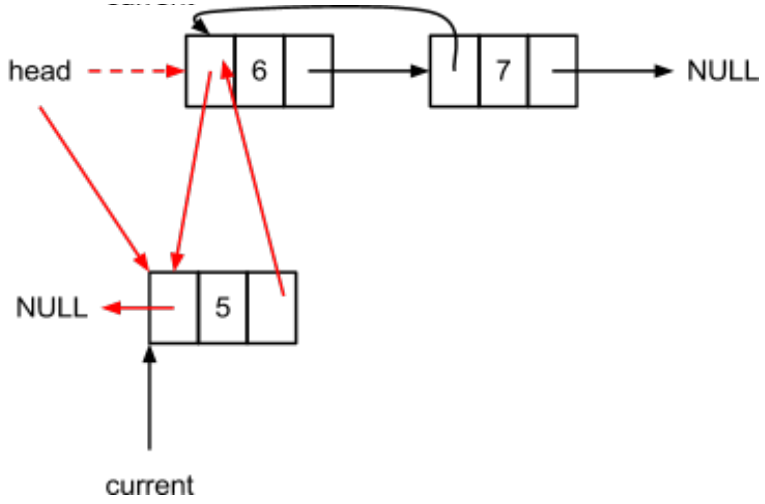
After insertion:



Case 2: The list is not empty.



After insertion:



Deletion from a Doubly Linked List

We want deletion to also be constant time operations. We are given pointers to the nodes we want to free.

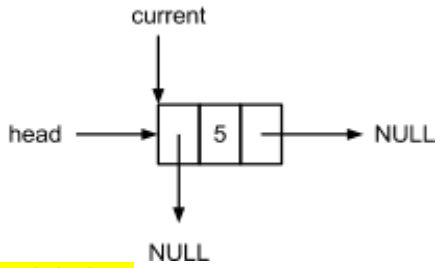
Question: When would we be removing nodes in the malloc pset? When we free nodes!

Question: What are the different cases we have for deletion?

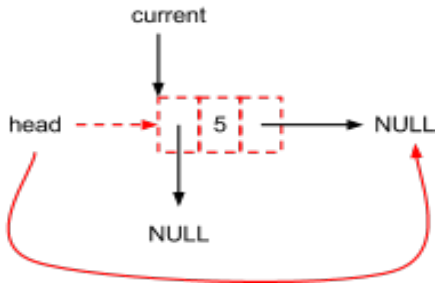
```
/** Remove @a current from the list pointed to by head.
 * @pre @a current must be an element of the list.
 * @post !contains_node(current) */
void remove_node(node *current) {
    // remove the node from our doubly linked list
    if (current->prev == NULL && current->next == NULL) {
        // case 1: it's the only item
        head = NULL;
```

```
    } else if (current->prev == NULL) {
        // case 2: it's the head of the list
        current->next->prev = NULL;
        head = current->next;
    } else if (current->next == NULL) {
        // case 3: it's the tail of the list
        current->prev->next = current->next;
    } else {
        // case 4: it's in the middle of the list
        current->prev->next = current->next;
        current->next->prev = current->prev;
    }
}
```

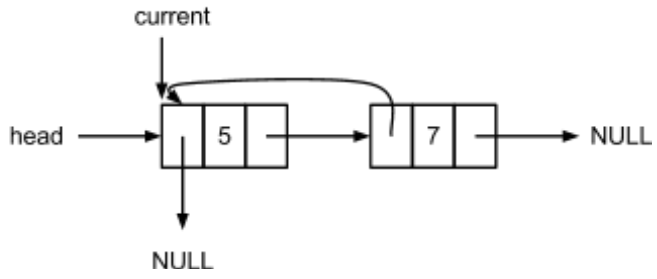
Case 1: The current element is the only element in the list.



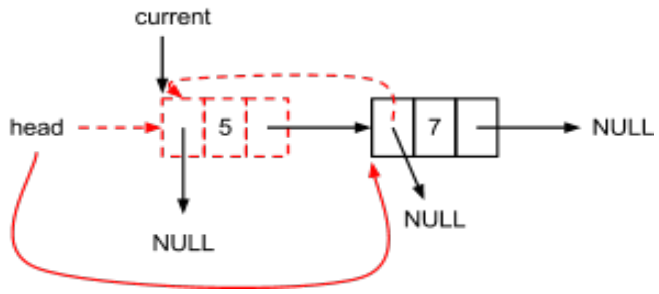
After deletion:



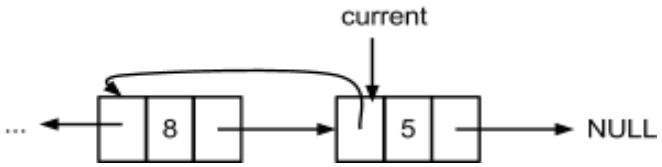
Case 2: The current element is the first element in the list (and not the last)



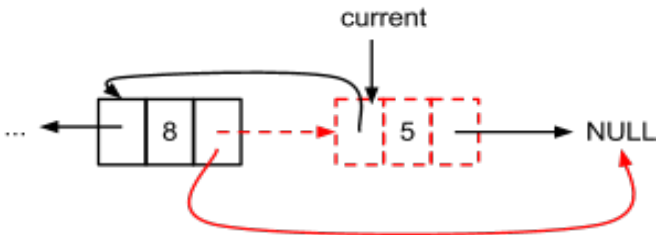
After deletion:



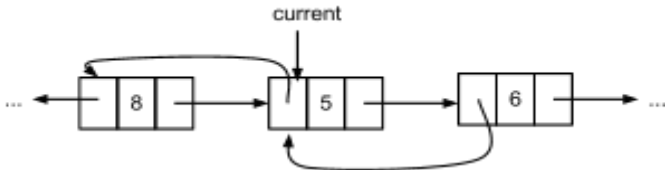
Case 3: The current element is the last element in the list (and not the first)



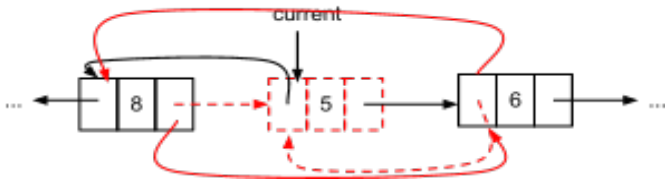
After deletion:



Case 4: The current element is in the middle of the list.



After deletion:



Checking if a node is in a Doubly Linked List

Question: How do we check if a node is in our list?

```
#include <stdbool.h>

/** Returns whether @a current is in the list pointed to by head. */
bool contains_node(node *current) {
    node *ptr = head;
    while (ptr != NULL) {
        if (ptr == current) {
            return true;
        }
        ptr = ptr->next;
    }
    return false;
}
```

4: Garbage Collection

Question: What is garbage collection? automatically reclaim memory in the heap

- this is a feature common in many languages
    - ex - Python, Java, ML
- There are some garbage collection tools in C, but they generally don't get all the garbage
    - GC in C is very **conservative**. We'll talk about why in a few minutes...
- How can we determine some block of memory is garbage?
    - let's create a root node in the stack or a global variable
    - memory blocks are **reachable** if there's a path from the root node, possibly through other pointers in the heap, then to the block. this means that the block is **live** and shouldn't be reclaimed.
    - but if there's no path... **garbage**!

Question: What kind of memory issues do garbage collectors solve?

- double free
- access after free
- memory leaks (sort of)

Question: How can we still have a memory leak with a garbage collector?

- If a memory block is **reachable** but never used, the GC will assume it's live, which can cause memory leaks.

- Before we mentioned GC in C in very conservative
    - In C, you can create pointers to anything:
        - `void *ptr = (void *) 0xCAFEBABE;`
    - all bit patterns in memory that could be pointers will be treated as such

Question: The variable `int i = 0xDEADBEEF`. This is just an integer value, but we have a conservative GC! Does anyone see a potential problem? Assume 0xDEADBEEF is a location somewhere in the heap...

- we'll have a false memory leak. the memory at location 0xDEADBEEF won't be reclaimed, even though we as programmers really only wanted to use that i as an integer!

- In C, you can end up creating pointers from ints by casting.
    - Other programming languages won't let you do this (and therefore their garbage collectors are much more powerful).
    - **tl;dr: Don't create pointers by casting random integers to pointers.**

Garbage Collection Techniques

Mark and Sweep

- set a mark in the header of each reachable memory block, starting from the root
    - do a full traversal of all the reachable nodes, and mark nodes that are still reachable (sweep step)
    - free any block without the mark bits set!

Question: What's the big disadvantage of this technique?

- GC will need to scan every object during each iteration.

Generation Garbage collection

- Most of the time, objects have a short lifespan.
- If an object has survived after a few GC cycles, it'll probably be around for a while longer.
- Thus, we can avoid iterating all the objects by using generations to separate them
    - If an object survives a GC sweep, move it into the next generation.
    - Run sweeps more frequently on younger generations, and less frequently on older ones
- Technique mostly used in Java

Reference counting

- Based on the idea that an object is garbage if its not reachable from the root
- If there are no references to an object, free it

Question: How should we decide if an object can be freed?

- Keep track of the references to the object. If there are none, free it!

Question: There's a problem with this approach... why?

- We have to update references on every iteration. Just like mark and sweep, this can become expensive!
- What do we do if we have a cycle of nodes, but these nodes are not reachable by any root pointer?
    - Their reference counts = 1, but they are unreachable.

5. Tips for Homework

0. Pointer Arithmetic.

See http://cs61.seas.harvard.edu/wiki/2012/ExerciseP for explanation and exercises.

1. Use structs to avoid complicated pointer arithmetic.

Suppose ptr is a pointer to the start of where we are placing our metadata, and we want to place the following types in the metadata: int, int, char*. Let's look at this code:

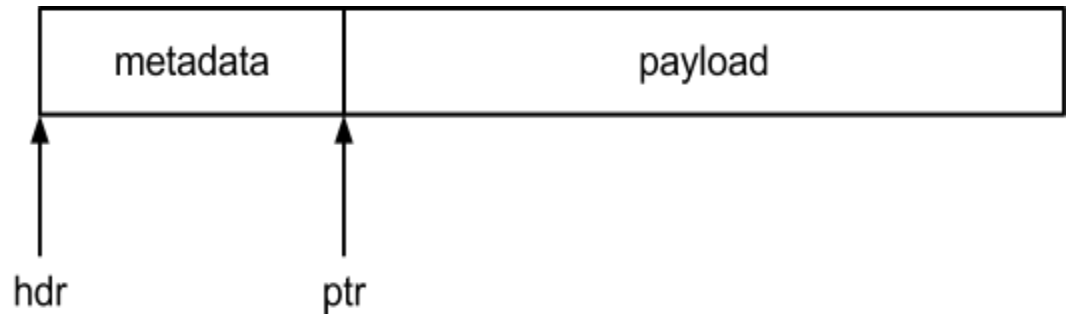
```
// declare metadata values
int a = 5;
int b = 6;
char *c = // some pointer

void *ptr = // pointer to beginning of metadata;
*((int *) ptr) = a;
*(((int *) ptr) + 1) = b;
*(((char **) ptr) + 2) = c; // because char * is 4 bytes on 32-bit machine
```

This is horrendous code. There is a much more elegant and robust solution using structs:

```
struct header {
  int a;
  int b;
  char *c;
};

void *ptr = // pointer to beginning of metadata;
struct header *hdr = (struct header *) ptr;
hdr->a = a;
hdr->b = b;
hdr->c = c;
```

2. Avoid pointer arithmetic on (void *) types!

Suppose ptr is a pointer to the beginning of the payload, and we want to move the pointer back to get to the beginning of the metadata:




```
void *ptr = // pointer to beginning of payload
```

```
struct header *hdr = (void *) (ptr - sizeof(struct header));
// don’t do this!!! void * arithmetic is not well defined
```

Instead:

```
void *ptr = // pointer to beginning of payload
struct header *hdr = ((struct header *) ptr) - 1; // much more robust
```

3. Use Git. Commit often and commit everything.

Not using git is like writing a paper for a humanities class and never saving the file.