Linked Lists

1320-Intermediate Programming

University of Texas at Arlington

Lecture Overview

- Quick Review
- Lecture
  - Memory-Stack vs Heap
  - Linked lists
- Sample Programs

QUICK REVIEW

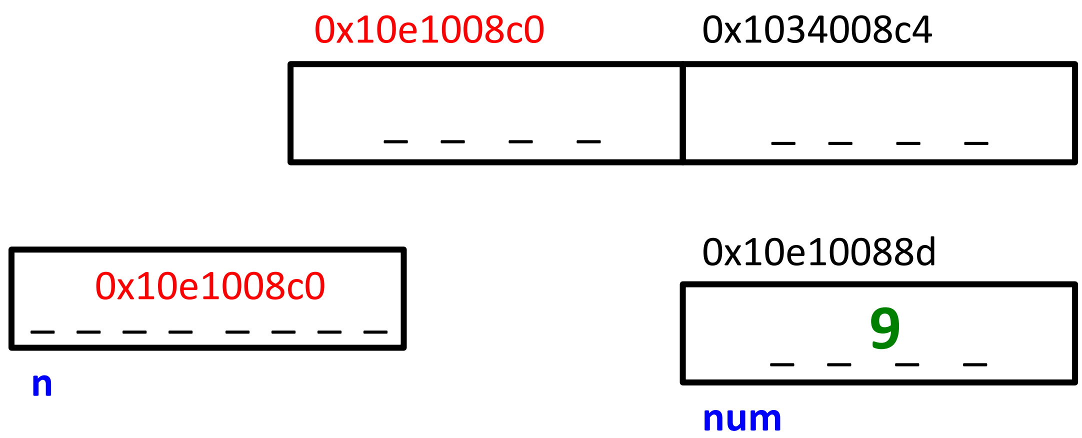
Memory Leaks

- So why is a memory leak bad?
  - Memory is a finite resource
  - If we keep allocating memory without releasing it, we are essentially hogging memory
- In worst case scenarios, it can crash your program or cause unwanted behavior

Example of a memory leak from overwriting your pointer:

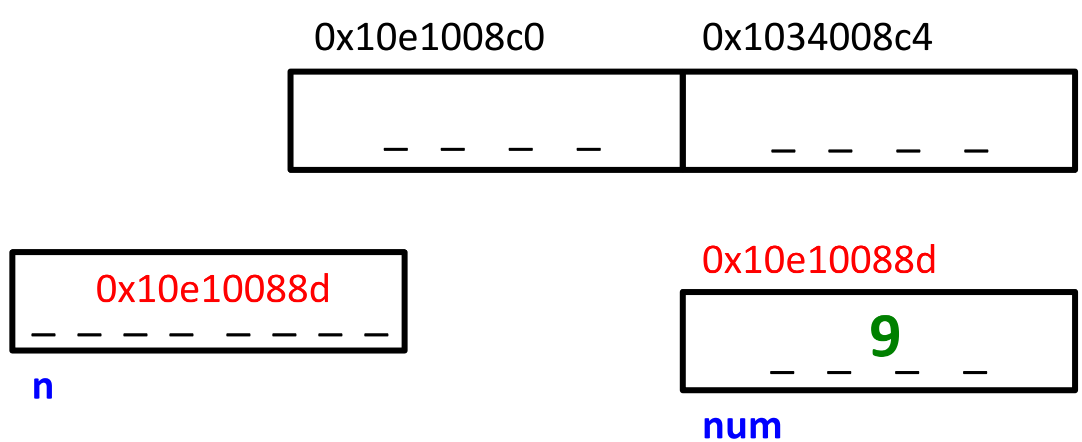
int *n=malloc(sizeof(int)*2)

int num=9;



The pointer n is pointing at our allocated memory.

n=#



The pointer n is now pointing at num and we don't have access to our allocated memory. (We should have freed it before we "lost" access to it)

**Note: allocated memory is a contiguous block*

LECTURE

Memory

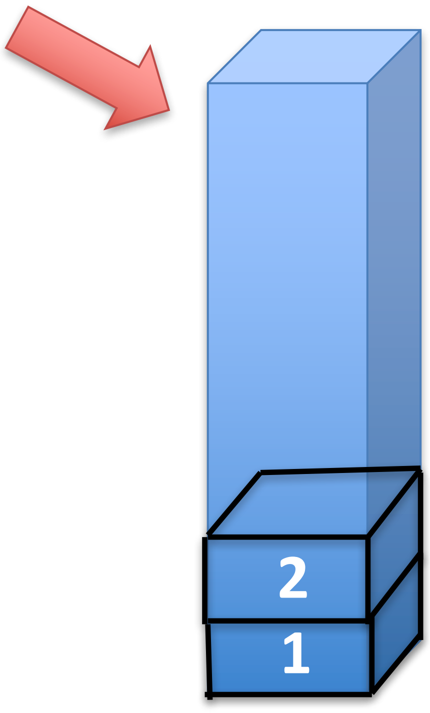
Linked Lists

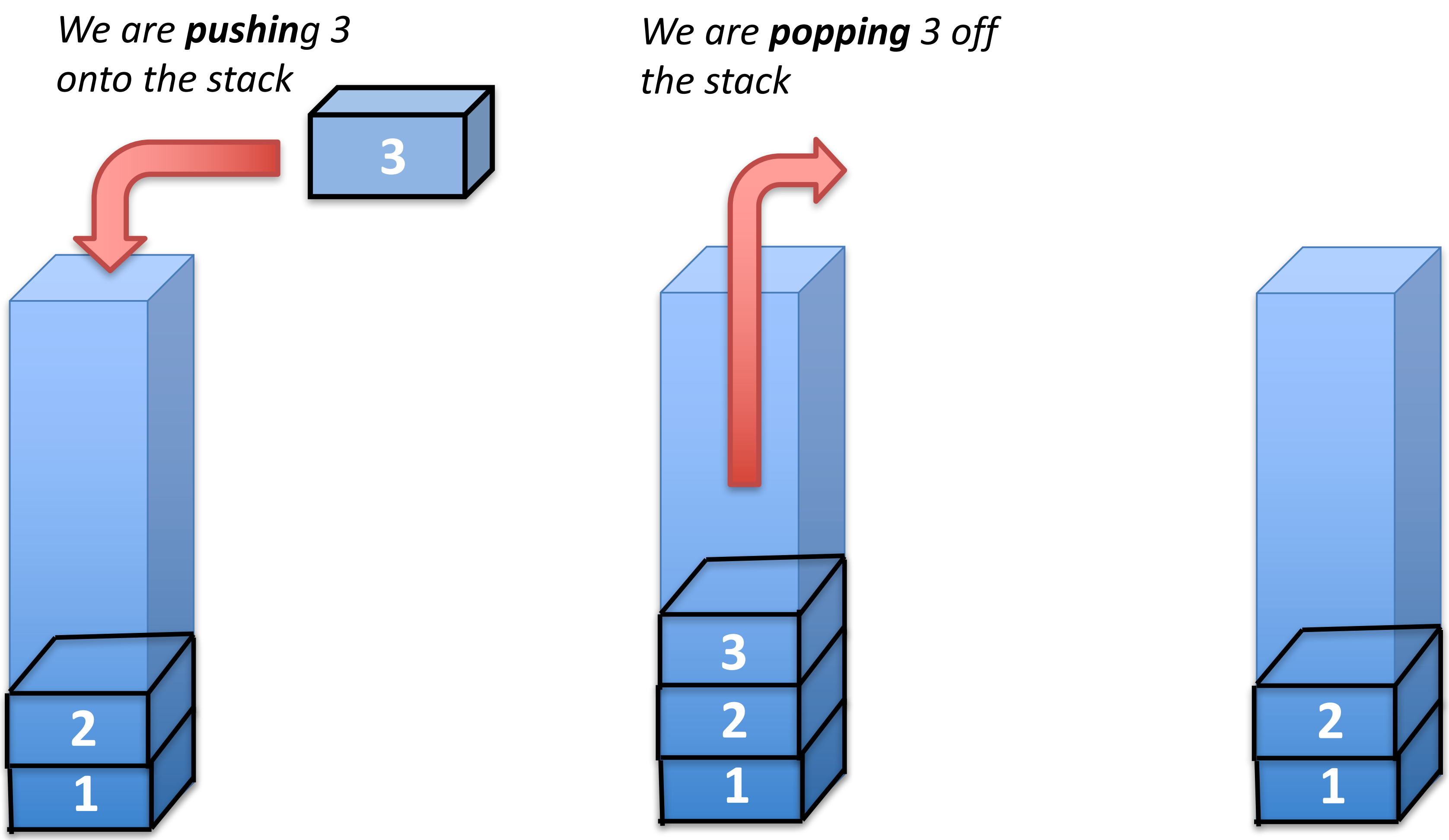
Memory

- Today I will give a **high level overview** of two important parts of memory that we use with our programs
  - Stack
  - Heap
- I will give general rules (there are exceptions to the rule) and high level details about memory

Stack Memory

- Before we discuss further, I want to mention that a **<u>stack</u>** is a type of data structure
- You can visualize it like this
- The only way to add data to the stack or delete data from the stack is from the **top** of the stack



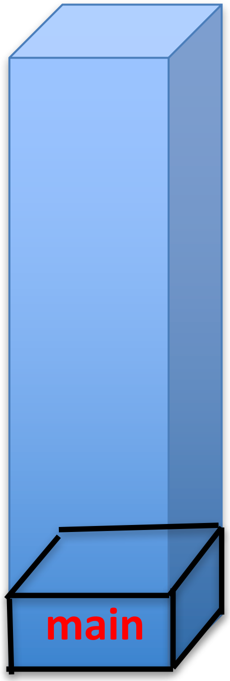


-Insertion (push) and deletion (pop) are ONLY allowed from the top of the stack
-Last in, first out (LIFO) (since 3 was the last in, it is the first out)
-First in, last out (FILO) (since 1 was the first in, it is the last out)

Stack Memory

- So what does this have to do with our programs?
- Every time our program is executed, it gets some memory to work with
  - We need to hold program information (like our variables) in memory
  - Our program has its own stack of memory allocated
    - This just means the memory is organized like a stack (previous slides)

Program Stack

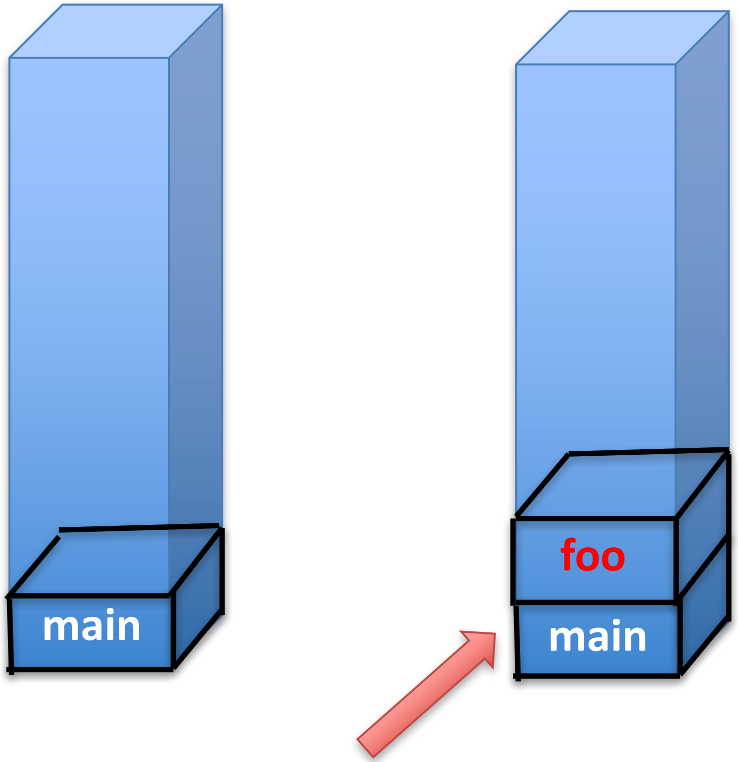


main goes onto the stack

```
#include <stdio.h>
void foo2()
{
        printf("Hi.");
}

void foo()
{
        foo2();
}

int main(int argc, char **argv)
{
        foo();
}
```

Note: the stack usually has a maximum size determined when your program starts

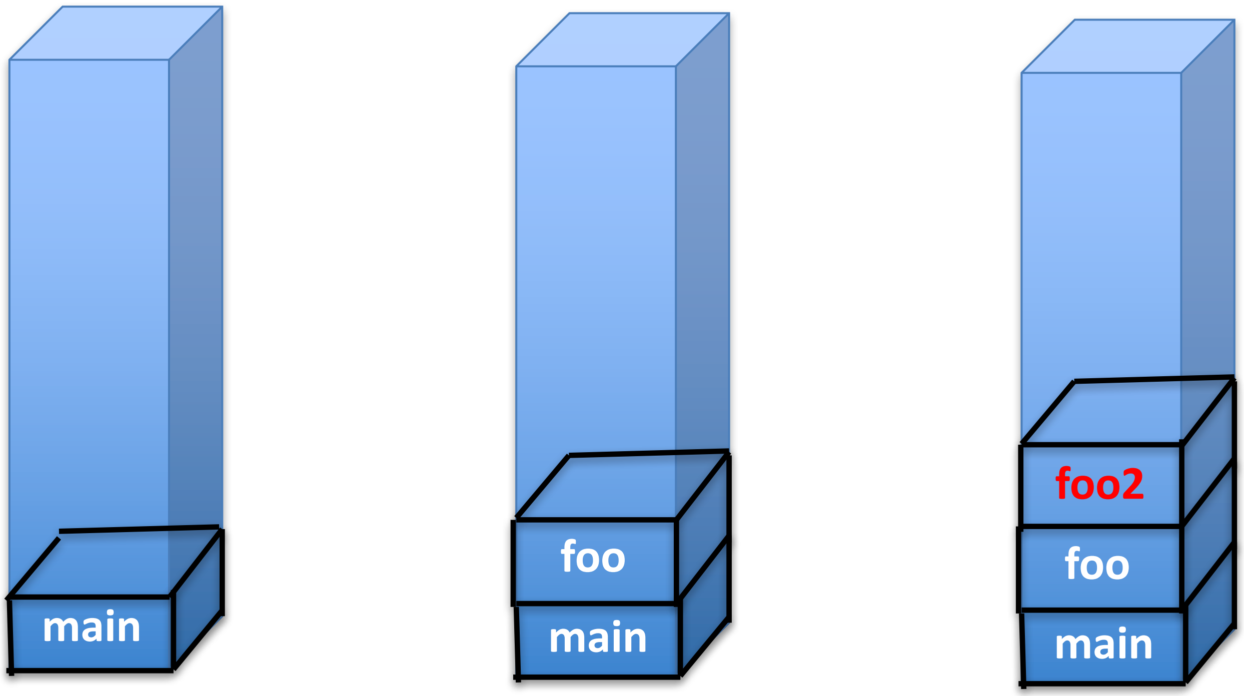
Program Stack



Note: Each of these is called a stack frame

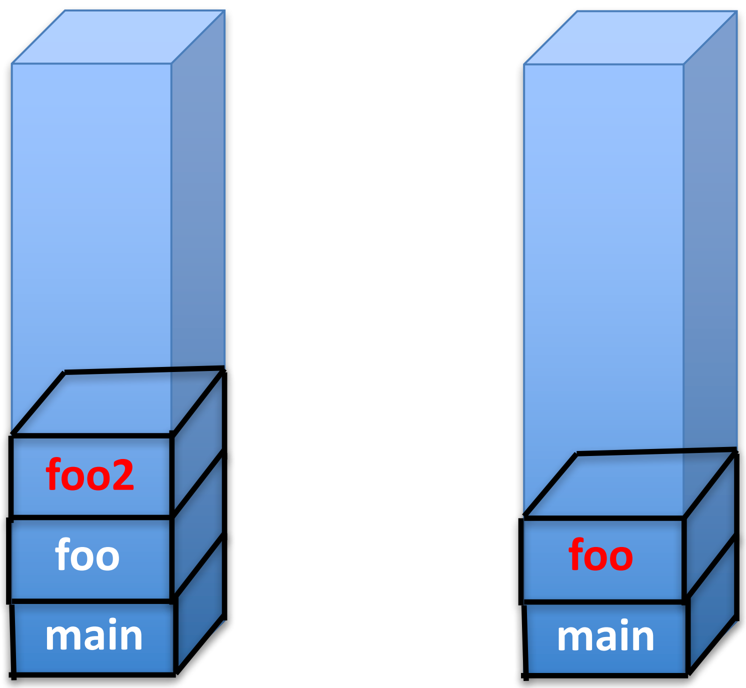
```
#include <stdio.h>
void foo2()
{
	printf("Hi.");
}

void foo()
{
	foo2();
}

int main(int argc, char **argv)
{
	foo();
}
```

Program Stack




```c
#include <stdio.h>
void foo2()
{
	printf("Hi.");
}

void foo()
{
	foo2();
}

int main(int argc....
{
	foo();
}
```

Program Stack

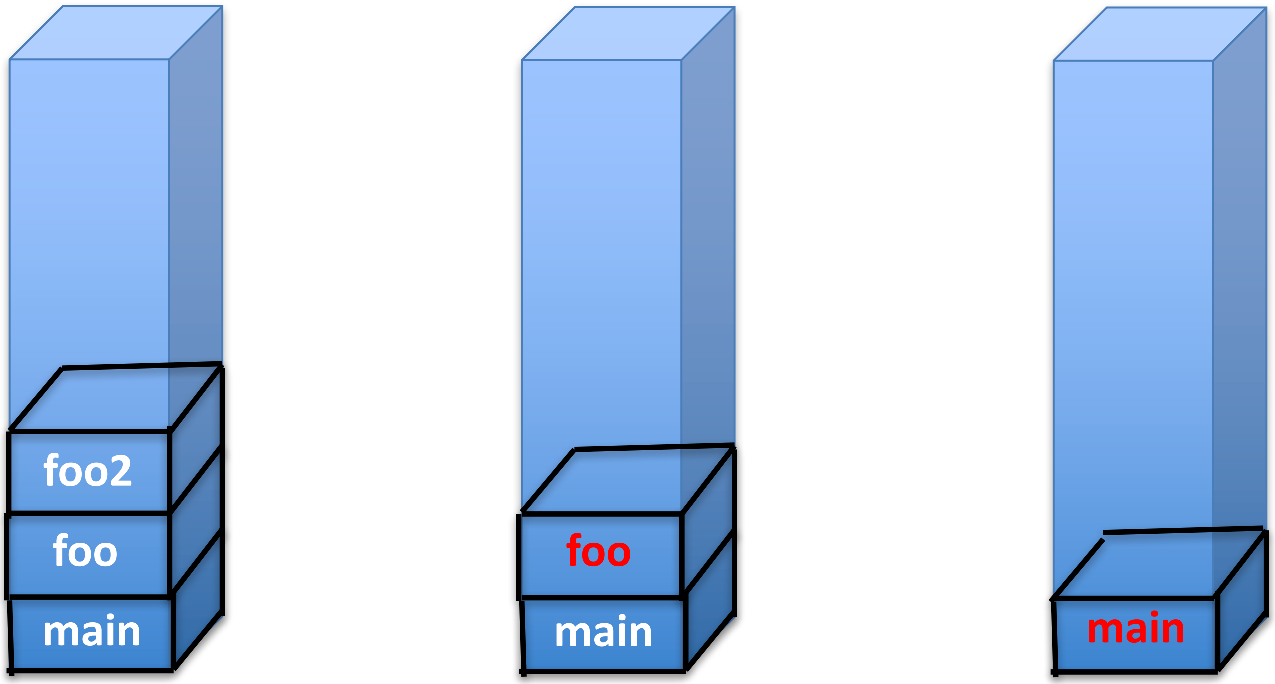



```c
#include <stdio.h>
void foo2()
{
	printf("Hi.");
}

void foo()
{
	foo2();
}

int main(int argc, char **argv)
{
	foo();
}
```

When foo2 is done executing, we go back to foo

Note: any variables created or used will automatically go out of scope and deallocate once a stack frame is done

Program Stack



**When foo is done executing,
we go back to main**

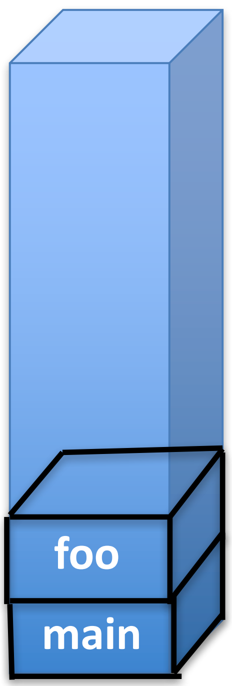
```
#include <stdio.h>
void foo2()
{
	printf("Hi.");
}

void foo()
{
	foo2();
}

int main(int argc …
{
	foo();
}
```

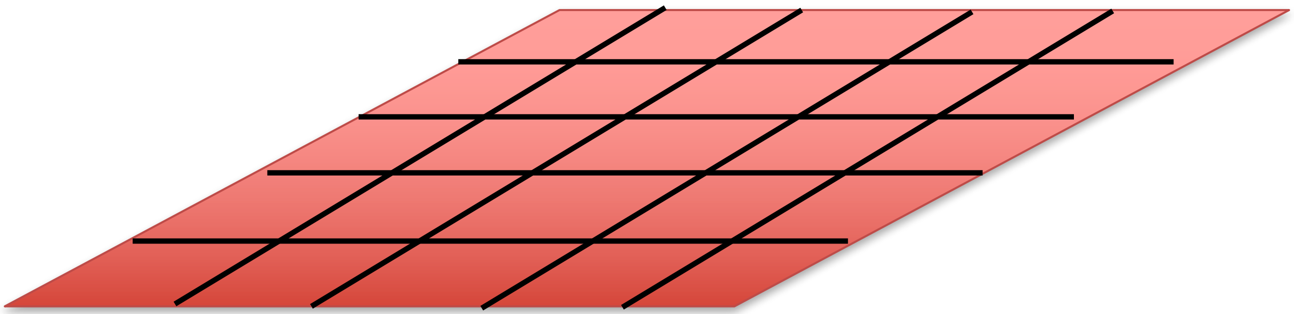
Heap Memory

- When we dynamically allocate our memory, we are taking memory from the heap
  - This is another area of memory for our program
- The pointer address we are holding in our return value from malloc is a heap address
  - The pointer itself (holding the address) is on the STACK
  - The address it is holding is an address on the HEAP
- The heap does **not** have a stack-like structure
  - It doesn’t follow the last in, first out rule

You can imagine the following:

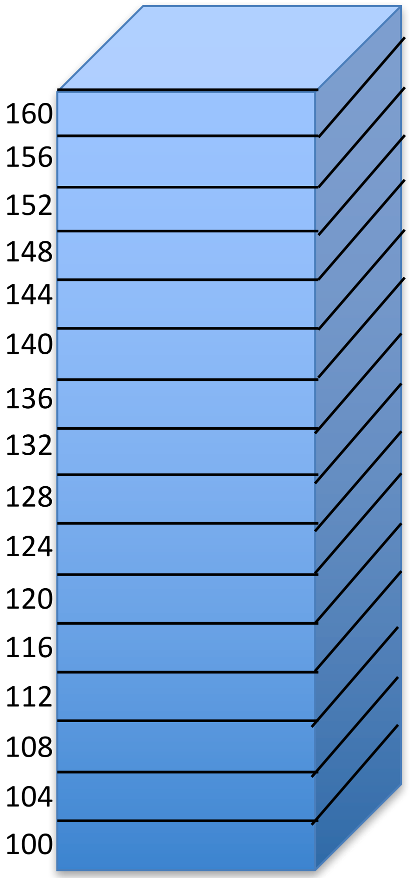
Note: this is a very high level explanation-many important details are not being mentioned



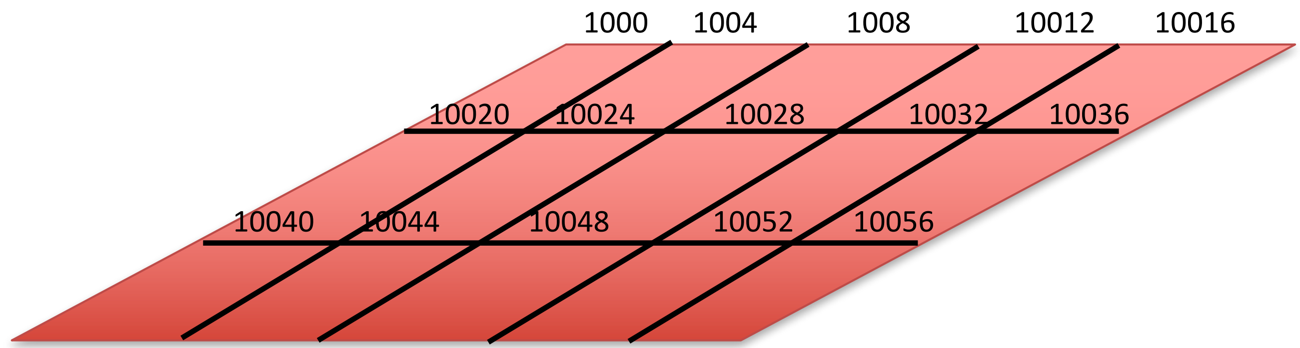


```c
#include <stdio.h>
#include <stdlib.h>

int sum(int*arr, int length)
{
	int i=0, answer=0;
	for(i=0;i<length;i++)
	{
		answer+=arr[i];
	}

	return answer;

}


int main(int argc, char **argv)
{
	int *a;
	int i;
	int length=12;
	int total;
	a=malloc(length*sizeof(int));
if(a==NULL)
	{
		printf("Memory not allocated.");
	}
	else
	{
		for(i=0;i<length;i++)
		{
			a[i]=i;
		}

		total=sum(a, length);
		printf("Total: %d\n", total);
		free(a);
	}


}
```

I will now do an example of this program in memory on the board (using the stack and heap-next slide).


160
156
152
148
144
140
136
132
128
124
120
116
112
108
104
100

Stack


1000
1004
1008
10012
10016
10020
10024
10028
10032
10036
10040
10044
10048
10052
10056

Heap

Memory

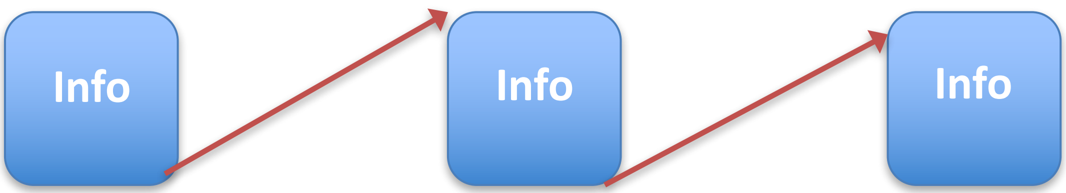
Linked Lists

Linked Lists

- A linked list is a basic data structure
  - It puts objects in some kind of linear order (but not necessarily in a physical order)
- It is not something “built” into the C language
  - Not like arrays or structs
- As computer scientists, we **implement** linked lists
  - We can implement it any language
  - In C, we will create a struct called Node

Linked Lists

This is the conceptual idea of a linked list:



(holds the address of the next node)

(holds the address of the next node)

The last node typically has a NULL (since there is no address for a next node)

We have a starting node (head)

We have a final node (tail)

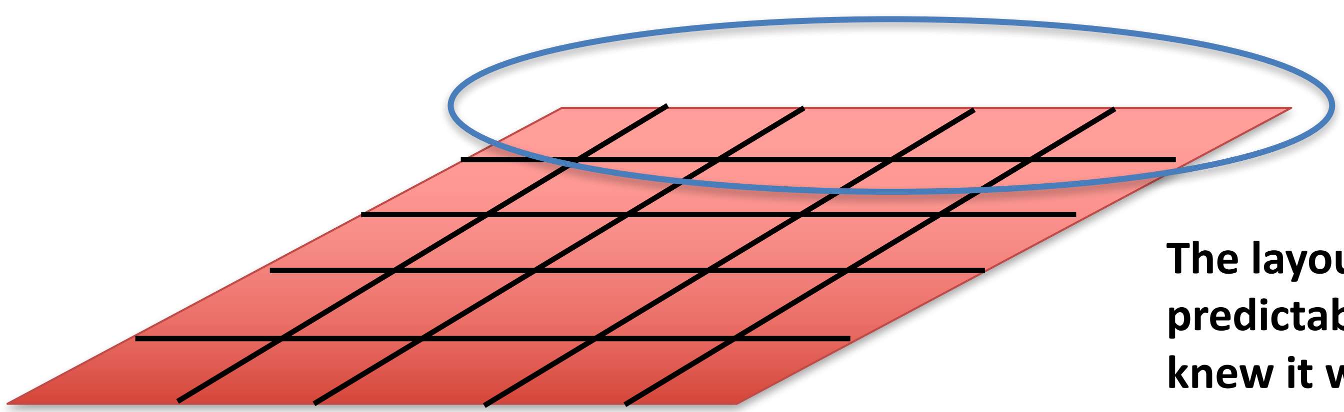
<u>Important notes:</u>

1) We can keep adding nodes (our list can grow) and deleting nodes (our list can shrink)- this allows for a flexible data structure

2) Nodes are not laid out contiguously in memory

Linked Lists

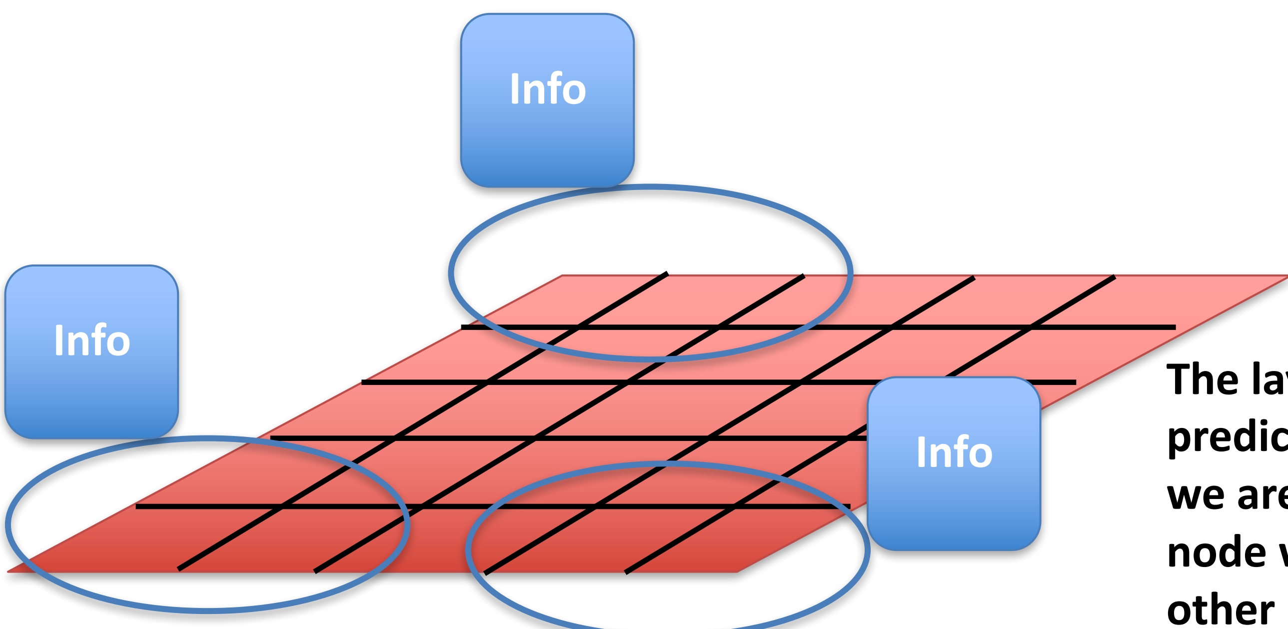
- Up until now, we have been dealing with contiguous blocks of memory (arrays and malloc)
  - We were able to use the index format [] and pointer arithmetic since we knew the layout



The layout of memory was predictable (because we knew it was contiguous)

Linked Lists

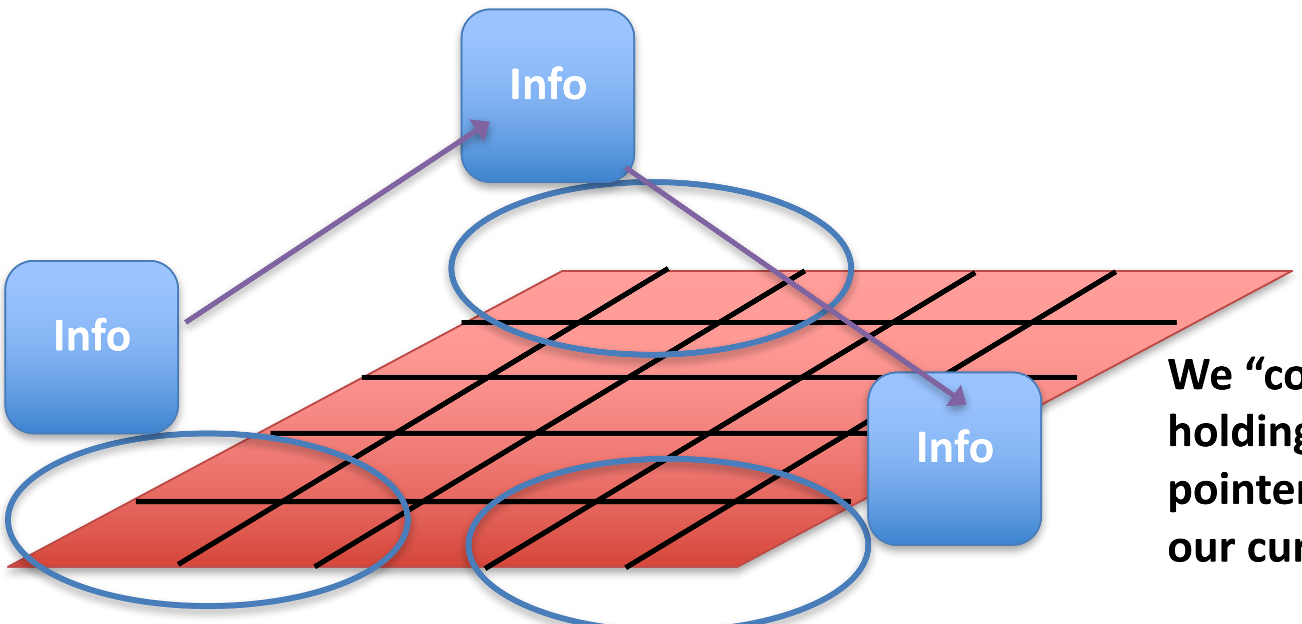
- While dealing with linked lists, we are not guaranteed this contiguous set up
  - We can't use the index method or increment a pointer to move along our linked list



The layout of memory is not predictable with linked lists- we are not guaranteed one node will be right after the other in memory

Linked Lists

- While dealing with linked lists, we are not guaranteed this contiguous set up
  - We can't use the index method or increment a pointer to move along our linked list



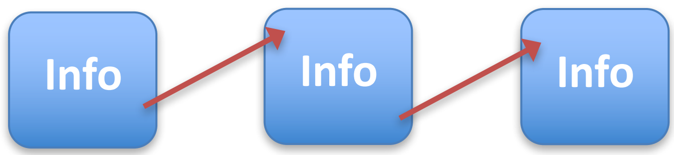
We "connect" our nodes by holding the address (using a pointer) of the next node in our current node

Linked Lists

- So how do we do the operations we are used to with data structures?
    - How do we traverse our data structure (move through it to print out information or look for a value)?
    - How do we update or change information in data structure?
- We need to understand the nature of the our data structure (and the nodes)
    - We will do an example today of traversing a linked list

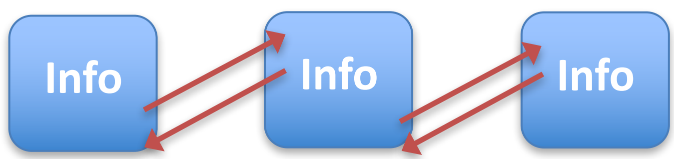
Linked Lists

- There are different types of linked lists
  - Common ones are:
    - Singly linked lists



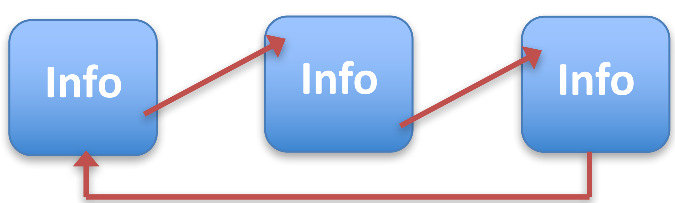
The node points to the next node in the list

    - Doubly linked lists



The node also points to the previous node

    - Circular linked lists



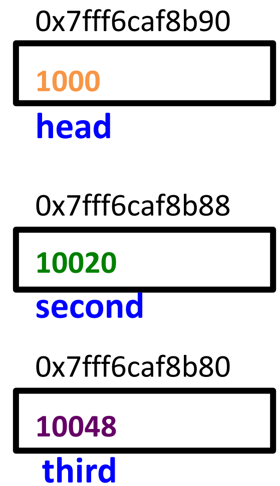
The last node points at the first node

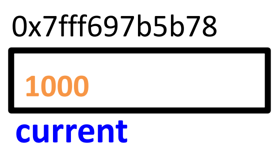
Linked Lists

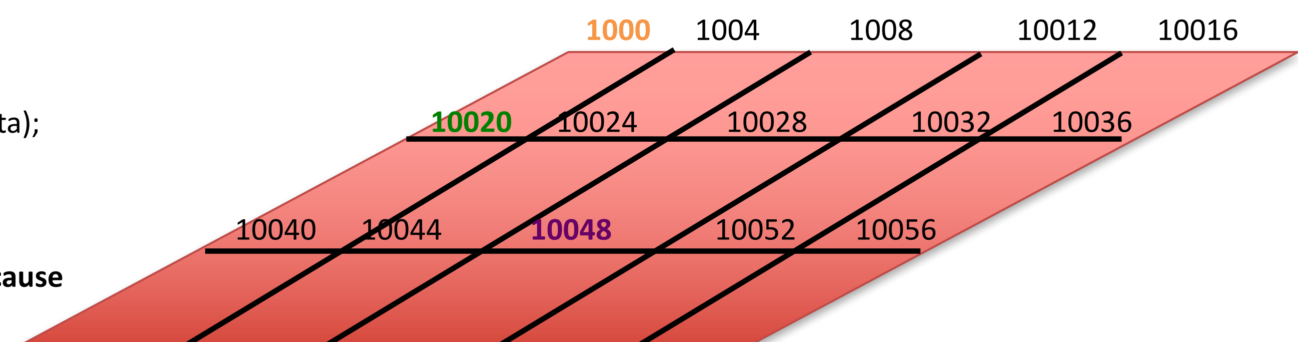
- Main operations on your linked list:
  - Traversal (going through your list)
  - Adding nodes
    - Where do you want to add the node?
  - Deleting nodes
    - Where do you want to delete the node?
  - Searching for a value

A visualization of your linked list in C

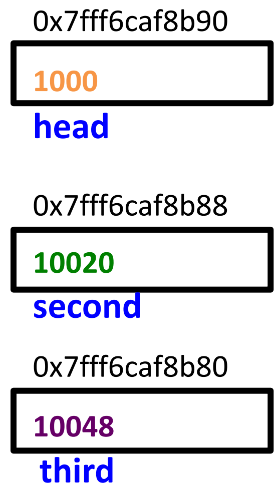
```
#include <stdio.h>
#include <stdlib.h>

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

int main(int argc, char **argv)
{
        Node* head  = malloc(sizeof(Node));
        Node* second = malloc(sizeof(Node));
        Node* third  = malloc(sizeof(Node));

        /*should make sure not NULL-saving space*/
        head->data=3;
        head->next=second;

        second->data=8;
        second->next=third;

        third->data=9;
        third->next=NULL;

        Node* current=head; /*start*/

        while(current!=NULL)
        {
                printf("Info: %d\n", current->data);
                current=current->next;
        }

        /*don't forget to free-I didn't here because
of space*/

}
```

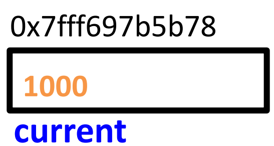




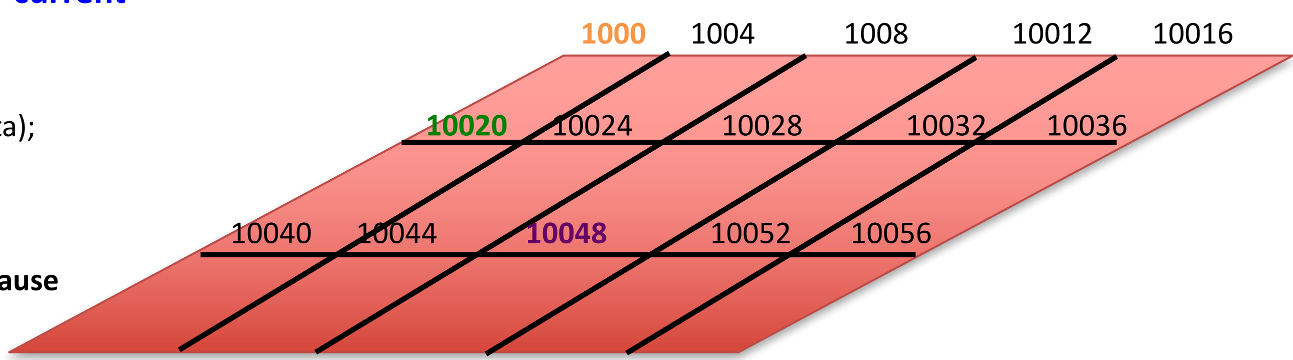


```
#include <stdio.h>
#include <stdlib.h>

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

int main(int argc, char **argv)
{
        Node* head  = malloc(sizeof(Node));
        Node* second = malloc(sizeof(Node));
        Node* third  = malloc(sizeof(Node));

        /*should make sure not NULL-saving space*/
        head->data=3;
        head->next=second;

        second->data=8;
        second->next=third;

        third->data=9;
        third->next=NULL;

        Node* current=head; /*start*/

        while(current!=NULL)
        {
                printf("Info: %d\n", current->data);
                current=current->next;
        }

        /*don't forget to free-I didn't here because
of space*/

}
```





The nodes are actually “linked up” by assigning the address of one node to the *next* member of another node



SAMPLE PROGRAM

Program

- I will give an example of a simple linked list (in future lectures, we will learn more)