

Quiz 0 Review Session

October 13th, 2014

Topics (non-exhaustive)

- Binary. ASCII. Algorithms. Pseudocode. Source code. Compiler. Object code. Scratch. Statements. Boolean expressions. Conditions. Loops. Variables. Functions. Arrays. Threads. Events.
- Linux. C. Compiling. Libraries. Types. Standard output.
- Casting. Imprecision. Switches. Scope. Strings. Arrays. Cryptography.
- Command-line arguments. Searching. Sorting. Bubble sort. Selection sort. Insertion sort. O . Ω . Θ . Recursion. Merge Sort.
- Stack. Debugging. File I/O. Hexadecimal. Strings. Pointers. Dynamic memory allocation.
- Heap. Buffer overflow. Linked lists.

Official Word

<http://cdn.cs50.net/2014/fall/quizzes/0/about0.html>

Tips for Quiz 0

- practice coding on paper (e.g., `strlen`, `atoi`)
- be familiar with your problem sets!
- do previous quizzes under time constraint
- creating your reference sheet is a great way to study

Data Types and Sizes

- char : 1 byte
- int : 4 bytes
- long long : 8 bytes
- float : 4 bytes
- double : 8 bytes
- <type>* : 4 bytes

Binary

conversion:

binary to decimal

$$101010_2 =$$

decimal to binary

$$50_{10} =$$

addition:

$$\begin{array}{r} 0 \ 1 \ 0 \ 0 \ 1 \\ + 1 \ 0 \ 0 \ 1 \ 1 \\ \hline \end{array}$$

Binary

conversion:

binary to decimal

$$101010_2 = 42$$

decimal to binary

$$50_{10} = 110010$$

addition:

$$\begin{array}{r} 0 1 0 0 1 \\ + 1 0 0 1 1 \\ \hline 1 1 1 0 0 \end{array}$$

Hexadecimal

conversion:

binary to hexadecimal

$11111111_2 =$

hexadecimal to binary

$0x5A =$

Hexadecimal

conversion:

binary to hexadecimal

$$11111111_2 = \text{0xFF}$$

hexadecimal to binary

$$\text{0x5A} = 01011010$$

ASCII - Math

Because characters are fundamentally just numbers, we can do math with chars!

```
int A = 65;
```

```
int B = 'A' + 1;
```

```
char C = 'D' - 1;
```

```
char D = 68;
```

```
printf("%c %c %c %c", A, B, C, D);
```

What will this print out?

ASCII - Math

Because characters are fundamentally just numbers, we can do math with chars!

```
int A = 65;
```

```
int B = 'A' + 1;
```

```
char C = 'D' - 1;
```

```
char D = 68;
```

```
printf("%c %c %c %c", A, B, C, D);
```

What will this print out? **A B C D**

Scope

Determines the region where a variable exists. Within this area, we can access or change the variable

- Global
 - Entire program has access to it
 - Exist for the duration of the program
- Local
 - Confined to a region
 - Examples: Within specific functions, if statements, for loops

What happens when we try to print at each point?

```
int global_variable = 0;

int main(void)
{
    global_variable = 5;
    if (global_variable == 5)
    {
        int local_variable = 3;
        printf("%i\n", local_variable);
    }
    printf("%i\n", local_variable);
    printf("%i\n", global_variable);
}
```

Prototypes

When we define a function after we plan to use it, we must include a prototype!

```
<return type> function_name(input list);
```

```
#include <stdio.h>

int cube(int input);

int main(void)
{
    int x = 2;
    printf("x is %i\n", x);
    x = cube(x);
    printf("x is %i\n", x);
}

int cube(int input)
{
    return input * input * input;
}
```

Floating Point Imprecision

infinitely many real numbers (even between 0 and 1!) but **finitely** many bits to represent real numbers

⇒ imprecision

Structs

Allow us to create our own data type or container to hold data of different types

```
typedef struct
{
    int id;
    string name;
}
student;
```

Creating and Accessing Structs

- Declare using the struct name as the variable type
- Access using the . operator

```
int main(void)
{
    student student_1;
    student_1.id = 1;
    student_1.name = "Daven";
}
```

Creating and Accessing Structs

- if we have a pointer to a struct we can use -> notation

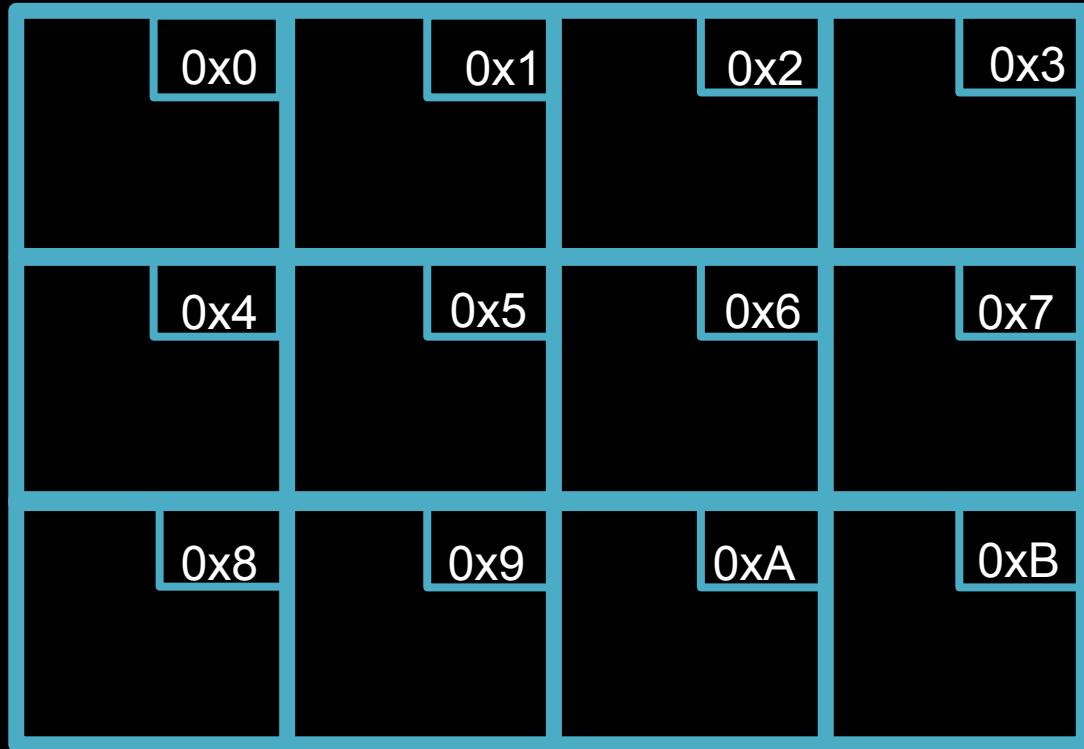
```
int main(void)
{
    student student_1;
    student* ptr = &student_1;

    ptr->name = "Rob";
    (*ptr).name = "Daven";
}
```

Pointers



Memory



Creating Pointers

`<type>*` `<variable name>`

Examples:

```
int* x;
```

```
char* y;
```

```
float* z;
```

Referencing and Dereferencing

Referencing:
&<variable name>

Dereferencing:
***<pointer name>**

Under the hood...

```
int x = 5;
```

```
int* ptr = &x;
```

```
int copy = *ptr;
```

Variable	Address	Value
x	0x04	5
ptr	0x08	0x04
copy	0x0C	5

Buggy

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

```
void to_five(int a)
```

```
{
```

```
  3:
```

```
    a = 5;
```

```
  4:
```

```
}
```

```
int main(void)
```

```
{
```

```
  1:
```

```
    int x = 3;
```

```
  2:
```

```
    to_five(x);
```

```
  5:
```

```
    printf("%d\n", x);
```

```
}
```

	x	a
1:		
2:		
3:		
4:		
5:		

Buggy

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

```
void to_five(int a)
```

```
{
```

```
  3:
```

```
    a = 5;
```

```
  4:
```

```
}
```

```
int main(void)
```

```
{
```

```
  1:
```

```
    int x = 3;
```

```
  2:
```

```
    to_five(x);
```

```
  5:
```

```
    printf("%d\n", x);
```

```
}
```

	x	a
1:	N/A	N/A
2:		
3:		
4:		
5:		

Buggy

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

```
void to_five(int a)
```

```
{
```

```
  3:
```

```
    a = 5;
```

```
  4:
```

```
}
```

```
int main(void)
```

```
{
```

```
  1:
```

```
    int x = 3;
```

```
  2:
```

```
    to_five(x);
```

```
  5:
```

```
    printf("%d\n", x);
```

```
}
```

	x	a
1:	N/A	N/A
2:	3	N/A
3:		
4:		
5:		

Buggy

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

```
void to_five(int a)
```

```
{
```

```
  3:
```

```
    a = 5;
```

```
  4:
```

```
}
```

```
int main(void)
```

```
{
```

```
  1:
```

```
    int x = 3;
```

```
  2:
```

```
    to_five(x);
```

```
  5:
```

```
    printf("%d\n", x);
```

```
}
```

	x	a
1:	N/A	N/A
2:	3	N/A
3:	3	3
4:		
5:		

Buggy

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

```
void to_five(int a)
```

```
{
```

```
  3:
```

```
    a = 5;
```

```
  4:
```

```
}
```

```
int main(void)
```

```
{
```

```
  1:
```

```
    int x = 3;
```

```
  2:
```

```
    to_five(x);
```

```
  5:
```

```
    printf("%d\n", x);
```

```
}
```

	x	a
1:	N/A	N/A
2:	3	N/A
3:	3	3
4:	3	5
5:		

Buggy

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

```
void to_five(int a)
```

```
{
```

```
  3:
```

```
    a = 5;
```

```
  4:
```

```
}
```

```
int main(void)
```

```
{
```

```
  1:
```

```
    int x = 3;
```

```
  2:
```

```
    to_five(x);
```

```
  5:
```

```
    printf("%d\n", x);
```

```
}
```

	x	a
1:	N/A	N/A
2:	3	N/A
3:	3	3
4:	3	5
5:	3	N/A

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

Fixed

Assume &x == 0x12

```
void to_five(int* a)
```

```
{
```

```
  3:
```

```
    *a = 5;
```

```
  4:
```

```
}
```

```
int main(void)
```

```
{
```

```
  1:
```

```
    int x = 3;
```

```
  2:
```

```
    to_five(&x);
```

```
  5:
```

```
    printf("%d\n", x);
```

```
}
```

	x	a	*a
1->			
2->			
3->			
4->			
5->			

Fixed

Assume &x == 0x12

```
#include <stdio.h>

void to_five(int* a)
{
    3:    *a = 5;
    4:
}

int main(void)
{
    1:    int x = 3;
    2:
    to_five(&x);
    5:
    printf("%d\n", x);
}
```

	x	a	*a
1->	N/A	N/A	N/A
2->			
3->			
4->			
5->			


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

Fixed

Assume &x == 0x12

```
void to_five(int* a)
```

```
{
```

```
  3:
```

```
    *a = 5;
```

```
  4:
```

```
}
```

```
int main(void)
```

```
{
```

```
  1:
```

```
    int x = 3;
```

```
  2:
```

```
    to_five(&x);
```

```
  5:
```

```
    printf("%d\n", x);
```

```
}
```

	x	a	*a
1->	N/A	N/A	N/A
2->	3	N/A	N/A
3->			
4->			
5->			

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

Fixed

Assume &x == 0x12

```
void to_five(int* a)
```

```
{
```

```
  3:
```

```
    *a = 5;
```

```
  4:
```

```
}
```

```
int main(void)
```

```
{
```

```
  1:
```

```
    int x = 3;
```

```
  2:
```

```
    to_five(&x);
```

```
  5:
```

```
    printf("%d\n", x);
```

```
}
```

	x	a	*a
1->	N/A	N/A	N/A
2->	3	N/A	N/A
3->	3	0x12	3
4->			
5->			

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

Fixed

Assume &x == 0x12

```
void to_five(int* a)
```

```
{
```

```
3:
```

```
    *a = 5;
```

```
4:
```

```
}
```

```
int main(void)
```

```
{
```

```
1:
```

```
    int x = 3;
```

```
2:
```

```
    to_five(&x);
```

```
5:
```

```
    printf("%d\n", x);
```

```
}
```

	x	a	*a
1->	N/A	N/A	N/A
2->	3	N/A	N/A
3->	3	0x12	3
4->	5	0x12	5
5->			

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

Fixed

Assume &x == 0x12

```
void to_five(int* a)
```

```
{
```

```
  3:
```

```
    *a = 5;
```

```
  4:
```

```
}
```

```
int main(void)
```

```
{
```

```
  1:
```

```
    int x = 3;
```

```
  2:
```

```
    to_five(&x);
```

```
  5:
```

```
    printf("%d\n", x);
```

```
}
```

	x	a	*a
1->	N/A	N/A	N/A
2->	3	N/A	N/A
3->	3	0x12	3
4->	5	0x12	5
5->	5	N/A	N/A


Pointer Arithmetic

Adding/subtracting **i** adjusts the pointer by **$i * \text{sizeof}(\langle \text{type of the pointer} \rangle)$** bytes

	x	y
<code>int x = 5;</code>	5	
<code>int* y = &x;</code>		
<code>y += 1;</code>		


Pointer Arithmetic

Adding/subtracting **i** adjusts the pointer by **i * sizeof(<type of the pointer>)** bytes

	x	y
int x = 5;	5	
int* y = &x;	5	0x04
y += 1;		

Pointer Arithmetic

Adding/subtracting **i** adjusts the pointer by **i * sizeof(<type of the pointer>)** bytes

	x	y
int x = 5;	5	
int* y = &x;	5	0x04
y += 1;	5	0x08

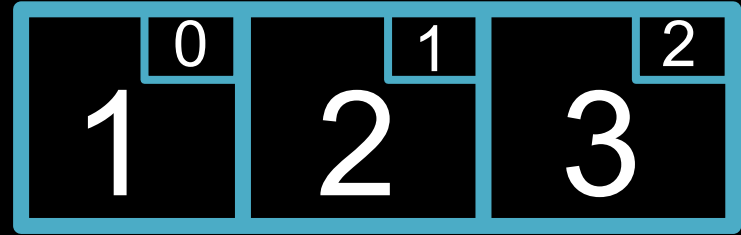
Pointers and Arrays

```
int array[3];
```

```
*array = 1;
```

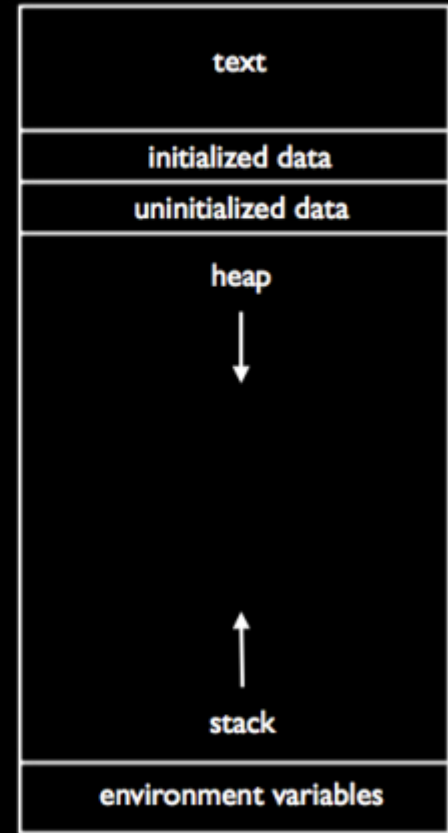
```
*(array + 1) = 2;
```

```
*(array + 2) = 3;
```

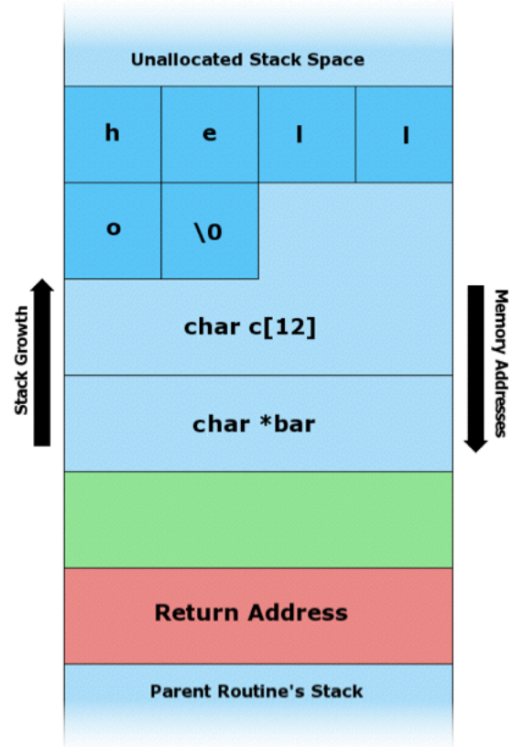


Memory

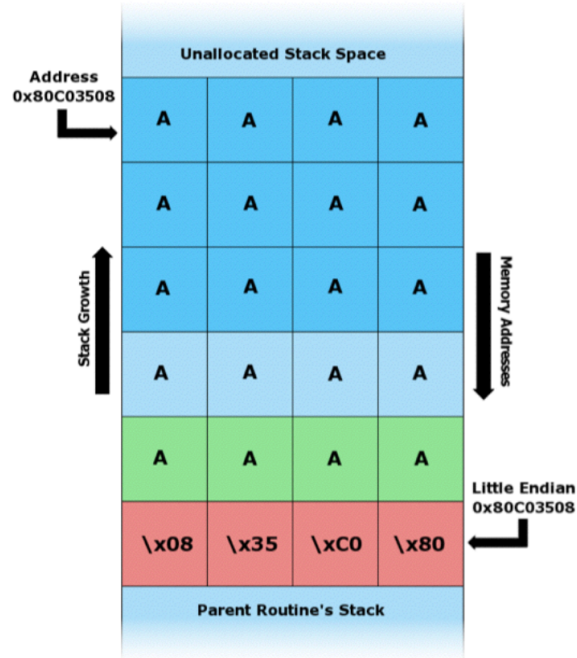
- **stack**: block of memory set aside when a program starts running
 - each function gets its own stack frame
 - **stack overflow**: when the stack runs out of space, results in a program crash and potential security concerns
- **heap**: region of unused memory that can be dynamically allocated using malloc
- don't forget to **free** dynamically allocated memory to prevent **memory leaks**



Buffer Overflow



Buffer Overflow



Allocating Memory

```
void* malloc(<size in bytes>;
```

```
    int* ptr = malloc(sizeof(int) * 10);
```

```
    ...
```

```
    free(ptr);
```

****Don't forget to check for NULL!****

Common Error Messages

- segmentation fault
 - when a program attempts to access memory that it is not allowed to access
 - check for NULL!
- implicit declaration of function
 - when a program is defined after the `main` function, and no **prototype** is present above
- undeclared identifier
 - when a variable has not been declared

Recursion

- a programming concept whereby a function invokes itself
- don't forget to include a base case!
- pros:
 - can lead to more concise, elegant code
 - some algorithms lend themselves to recursion
 - e.g. merge sort

Search and Sort Run Times

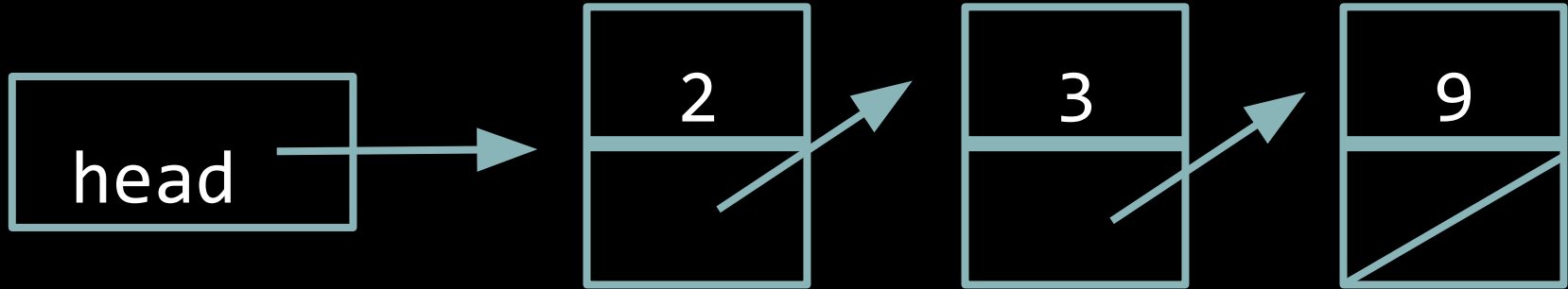
	linear search	binary search	bubble sort	selection sort	insertion sort	merge sort
O	n	$\log(n)$	n^2	n^2	n^2	$n \log(n)$
Ω	1	1	n	n^2	n	$n \log(n)$
Θ				n^2		$n \log(n)$

O upper bound (in the worst case)

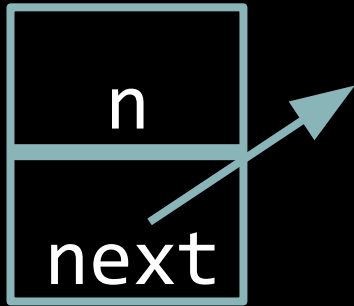
Ω lower bound (in the best case)

Θ identical upper and lower bound (in both cases)

Linked Lists

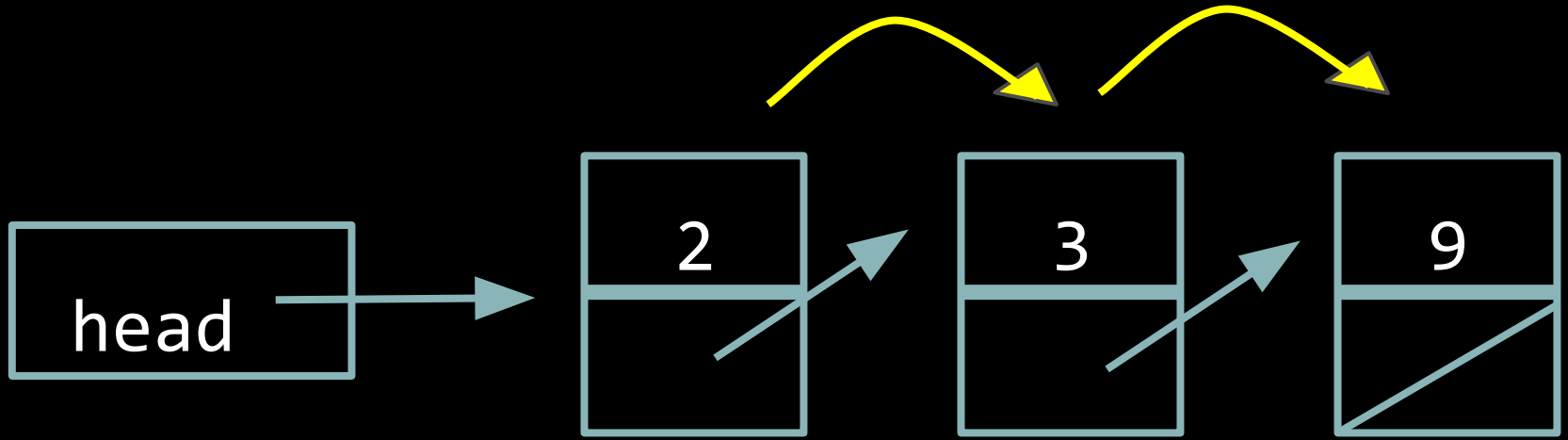


Nodes



```
typedef struct node
{
    int n;
    struct node* next;
}
node;
```

Search

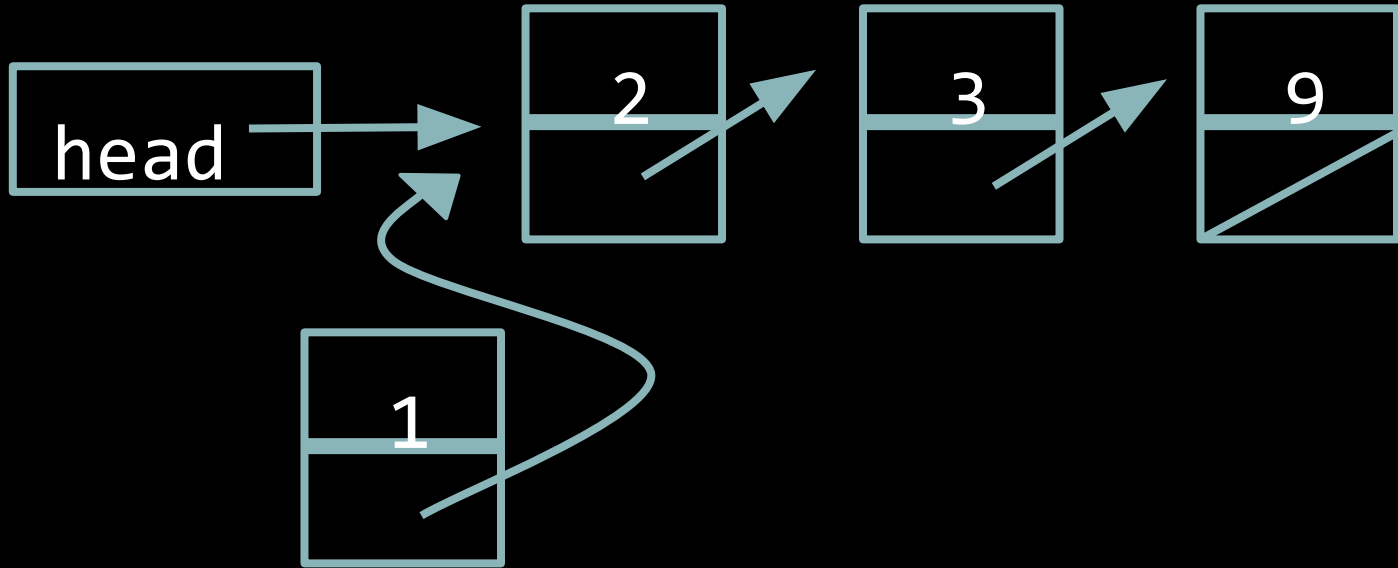


```
bool search(node* list, int n)
{
    // points at current node
    node* ptr = list;

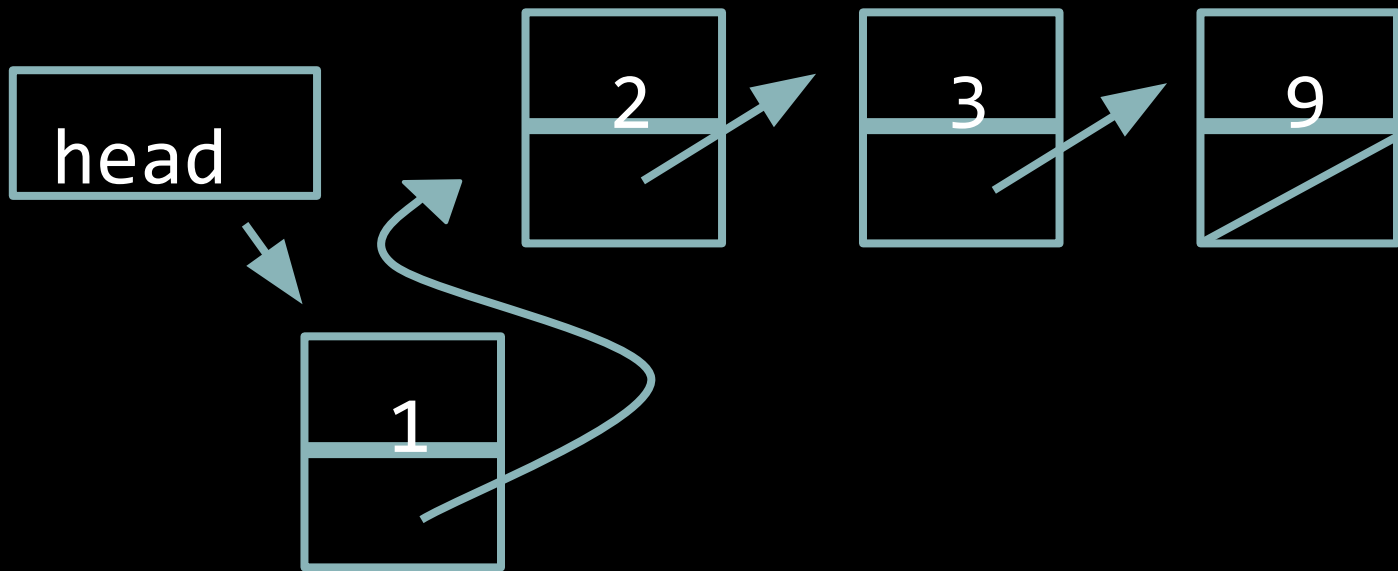
    // traverse the list until the end
    while (ptr != NULL)
    {
        // check if we found value
        if (ptr->n == n)
        {
            return true;
        }

        // move on to next element
        ptr = ptr->next;
    }
    return false;
}
```

Insertion



Insertion



```
void insert(int n)
{
    // create new node
    node* new = malloc(sizeof(node));

    // check for NULL
    if (new == NULL)
    {
        exit(1);
    }

    // initialize new node
    new->n = n;
    new->next = NULL;

    // insert new node at head
    new->next = head;
    head = new;
}
```

GDB

- allows you to execute a program line by line and see what is going on inside it
- a few important commands:
 - break
 - run
 - next
 - step
 - print
 - continue

Who needs a computer?

Code by hand:

- `strlen()`
- `atoi()`
- `toupper()`
- `pow()`
- `GetPositiveInt()`
- `RandomInt()`

Questions?

And finally...

RELAX AND SLEEP!

(you'll do great! =D)