CSE 3100: Systems Programming

Lecture 8 : Linked Lists, Enums and Function Pointers

Department of Computer Science and Engineering

University of Connecticut

Today's Lecture Topics

1. Linked Lists and Structures

2. Enumeration Types and Constants

3. Function Pointers

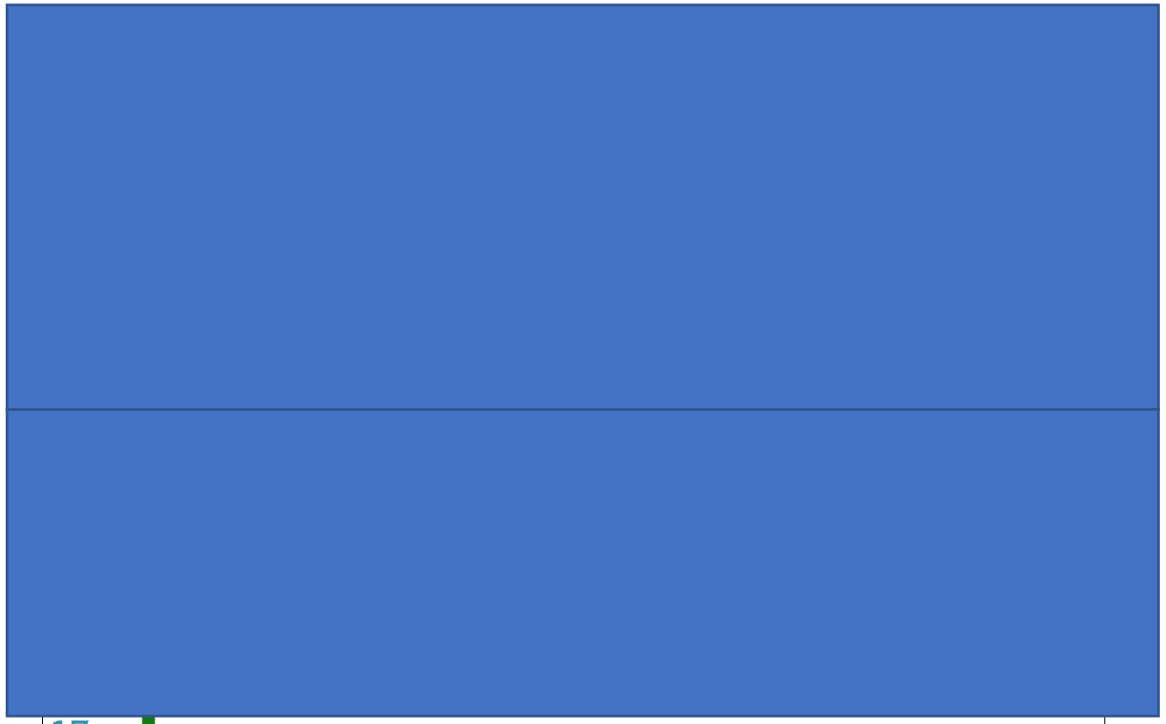
1. Linked Lists and Structures

Review from last lecture…structures

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

typedef struct studentDataStruct {
    int    midtermGrade; // grade data
    int    homeworkGrade; //grade data 2
    char ID[6]; //6 Character student ID
} sData; // Define a type. Easier to use.
```

Here we define a student structure and some related variables

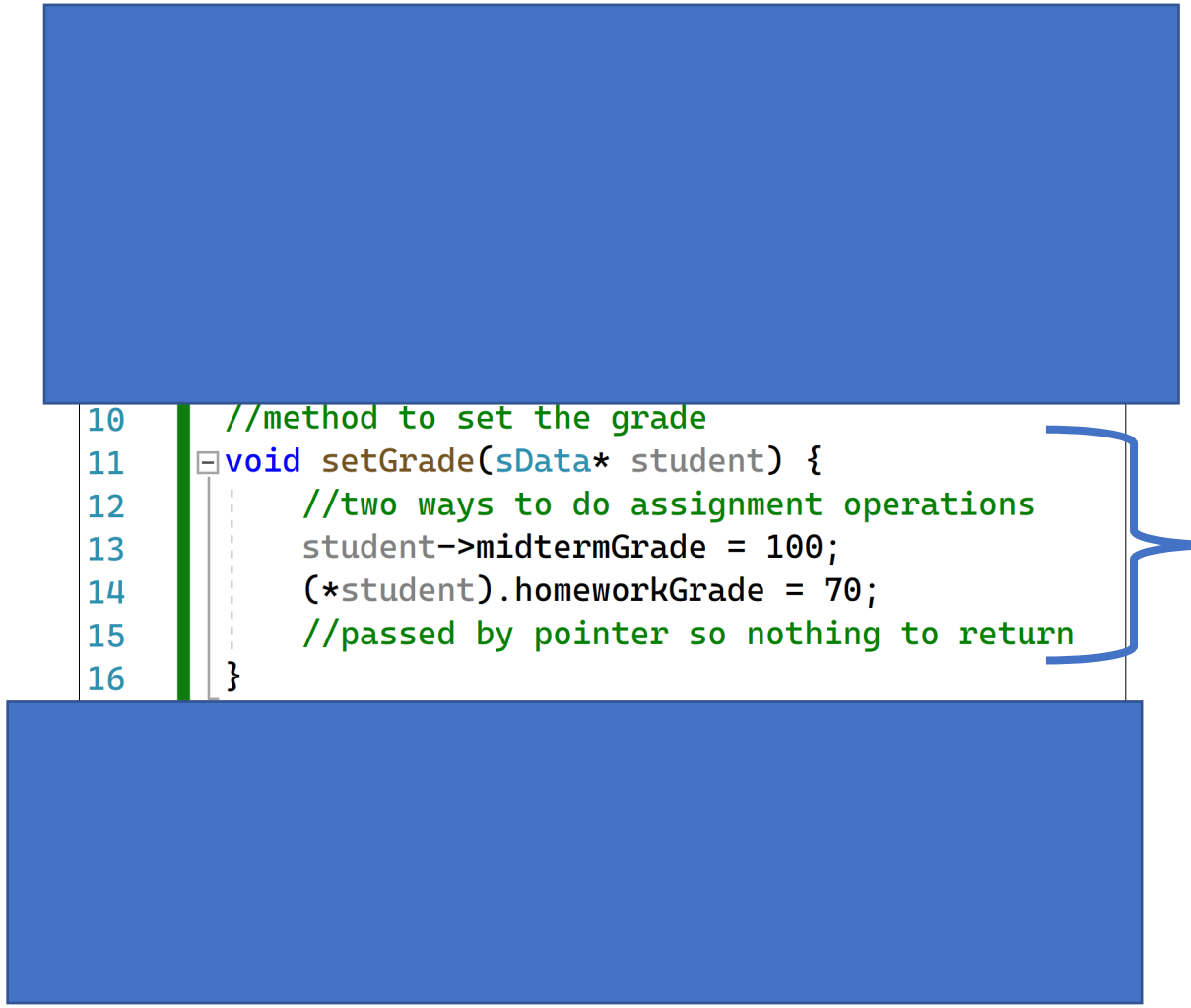
Structures Example



```
17
18  void main() {
19      sData studentA; //create a student struct
20      //create a pointer to the student struct
21      sData* sPointer = &studentA;
22      setGrade(sPointer);
23  }
```

In main create an instance of the student structure and call the set grades function.

Structures Example



```
10  //method to set the grade
11  void setGrade(sData* student) {
12      //two ways to do assignment operations
13      student->midtermGrade = 100;
14      (*student).homeworkGrade = 70;
15      //passed by pointer so nothing to return
16  }
```

- Here we have the set grades function where we take a pointer to a student object.
- Note how C allows TWO different (but operationally equal) ways to set the variables of a structure when only a pointer is given.

Structures Example Entire Code

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

typedef struct studentDataStruct {
    int    midtermGrade; // grade data
    int    homeworkGrade; //grade data 2
    char ID[6]; //6 Character student ID
} sData; // Define a type. Easier to use.

 //method to set the grade
void setGrade(sData* student) {
    //two ways to do assignment operations
    student->midtermGrade = 100;
    (*student).homeworkGrade = 70;
    //passed by pointer so nothing to return
}

void main() {
    sData studentA; //create a student struct
    //create a pointer to the student struct
    sData* sPointer = &studentA;
    setGrade(sPointer);
}
```

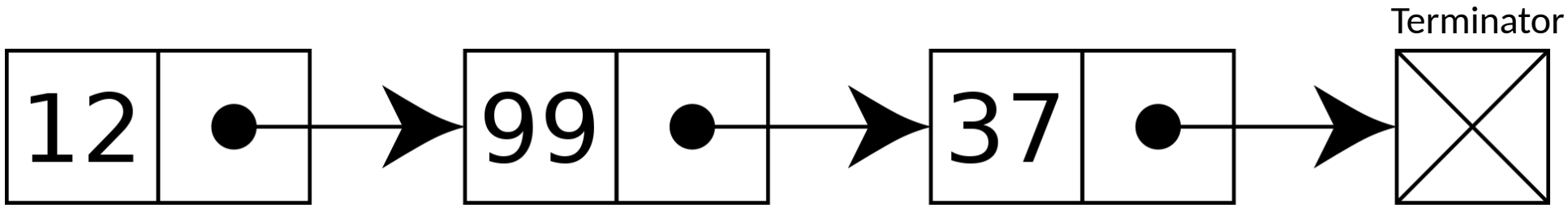
Code to define a structure

Function that operates on a pointer to a structure

Main where we instantiate the structure

Structures in C in practice: Linked Lists

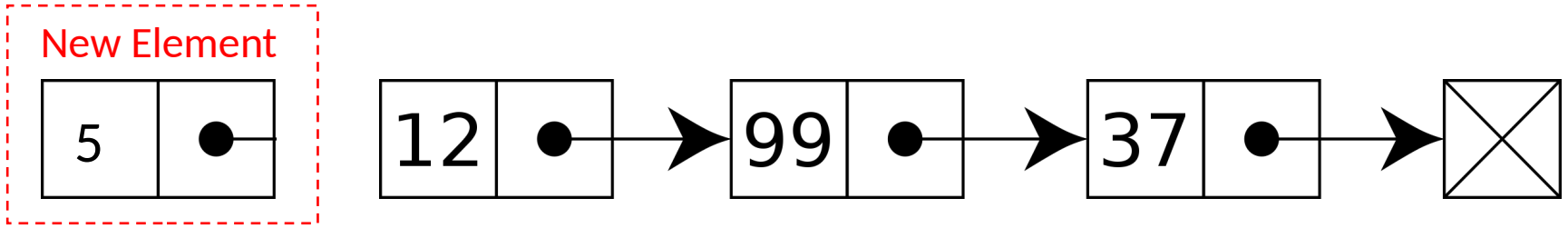
Singly Linked Lists



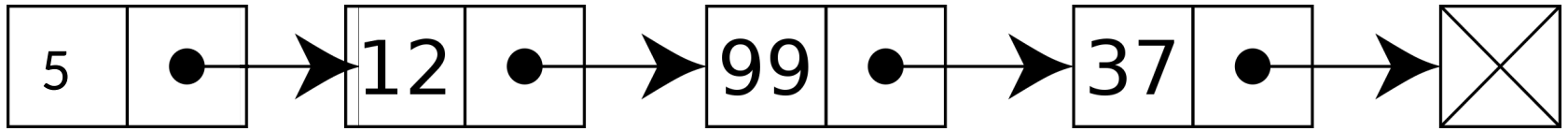
- Linear collection of data elements as nodes.
- Each node contains a value and a pointer to the next element.
- The last element points to a terminator to show that the end of the list has been reached.

Figure Source: https://upload.wikimedia.org/wikipedia/commons/thumb/6/6d/Singly-linked-list.svg/1920px-Singly-linked-list.svg.png

Simple Single Linked List Example

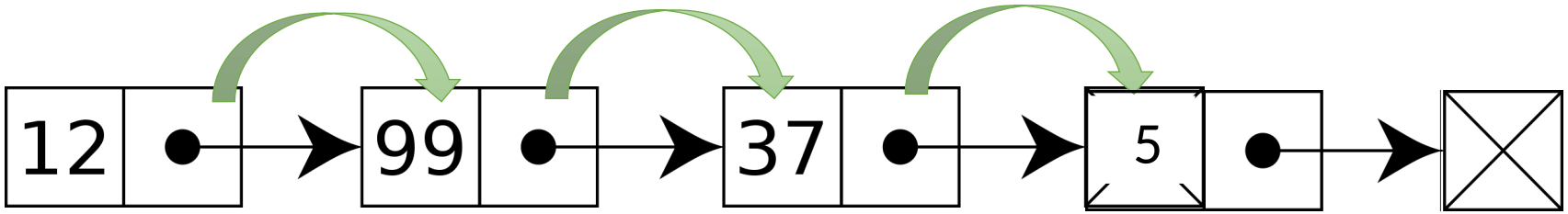


Inserting at the head of the list is easy! What is the time complexity?



O(1)

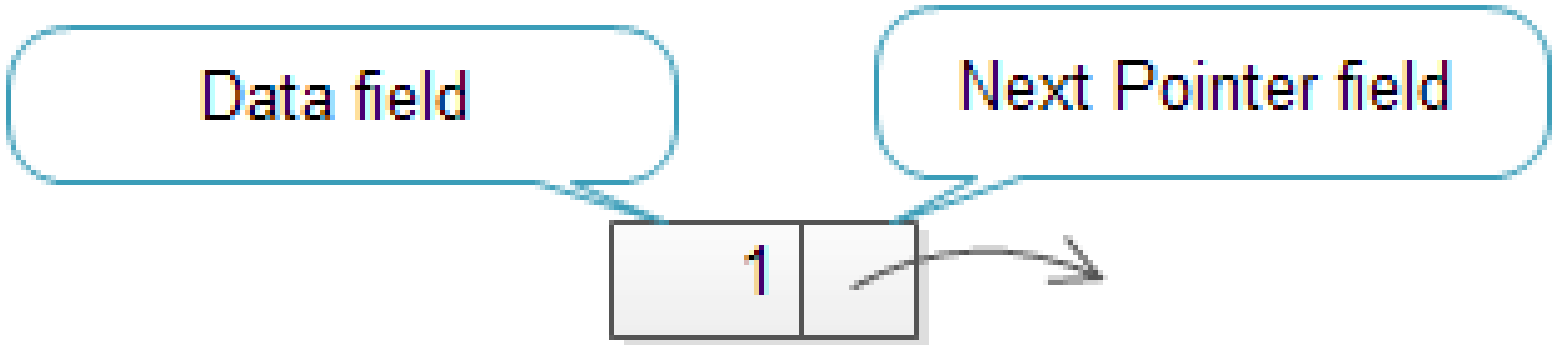
Now I want to insert at an arbitrary location, e.g. after node value 37. What is the time complexity?



O(n)

Linked Lists

```
// A data structure that consists of a chain of nodes
// Starting from head, a node has a reference to the next node


typedef struct node_tag {
   int    v;   // data
   struct node_tag * next;    // A pointer to this type of struct
} node;           // Define a type. Easier to use.
```



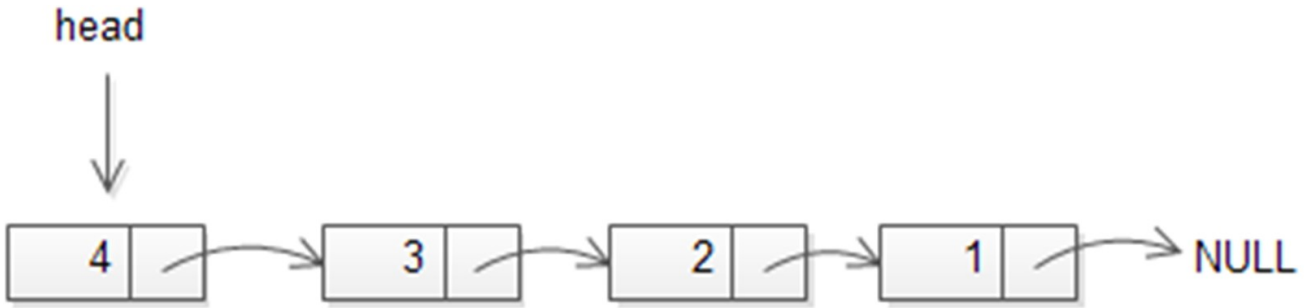
Linked List Head

Function: Removes first item from a linked list.

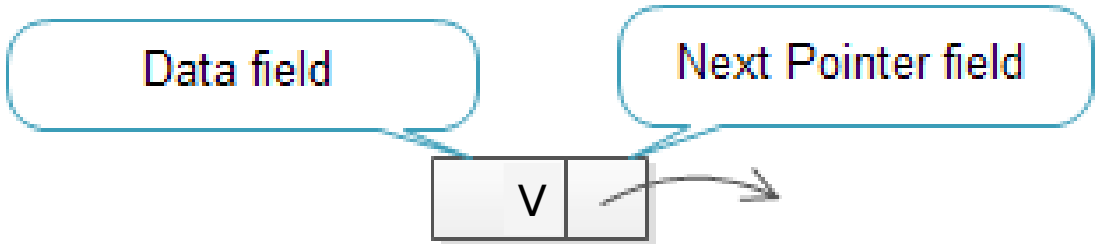
Second Item:


```
node * head;      // head is a pointer, not a node!


head = NULL;  // at beginning, it is empty
```

After adding nodes into the list:



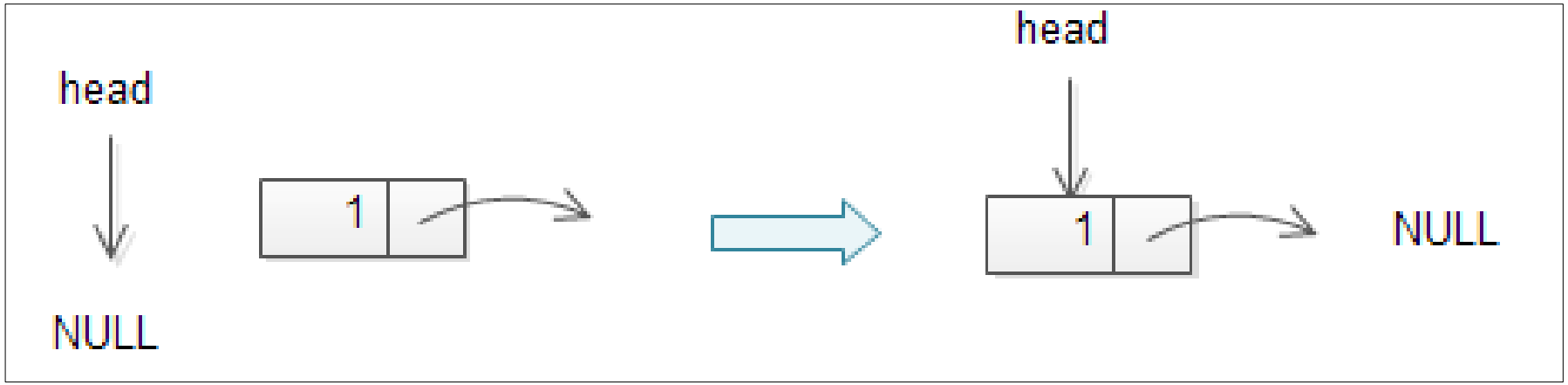
Creating a Node



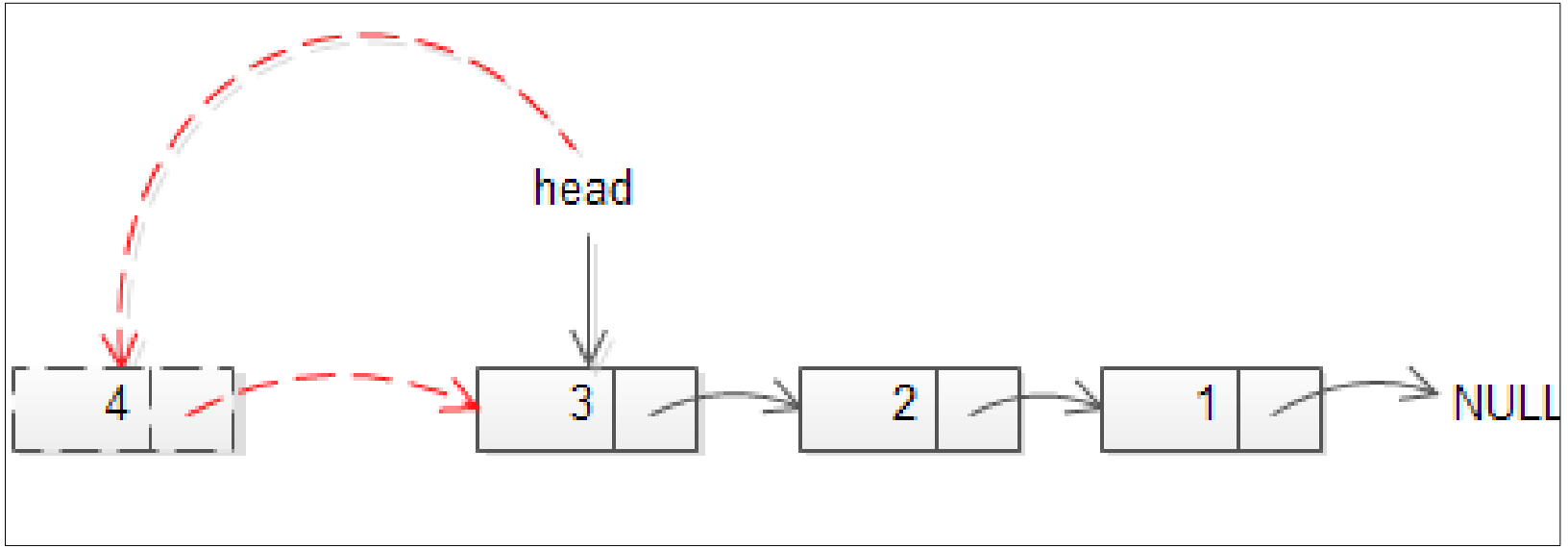

```
node * new_node(int v)      // create a node for value v
{
    node * p = malloc(sizeof(node)); // Allocate memory
    assert(p != NULL);    // you can be nicer

    // Set the value in the node.
    p->v = v;     // you could do (*p).v
    p->next = NULL;
    return p;     // return
}
```

Adding a new node to a linked list (Prepend)

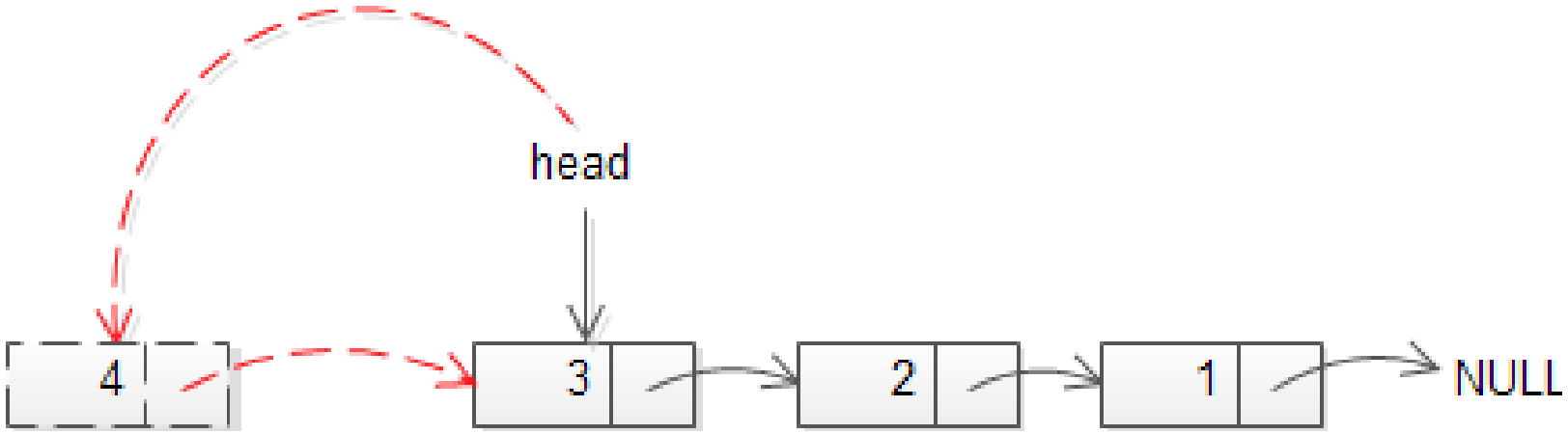
When the linked list starts as empty:



When you want to replace the head:



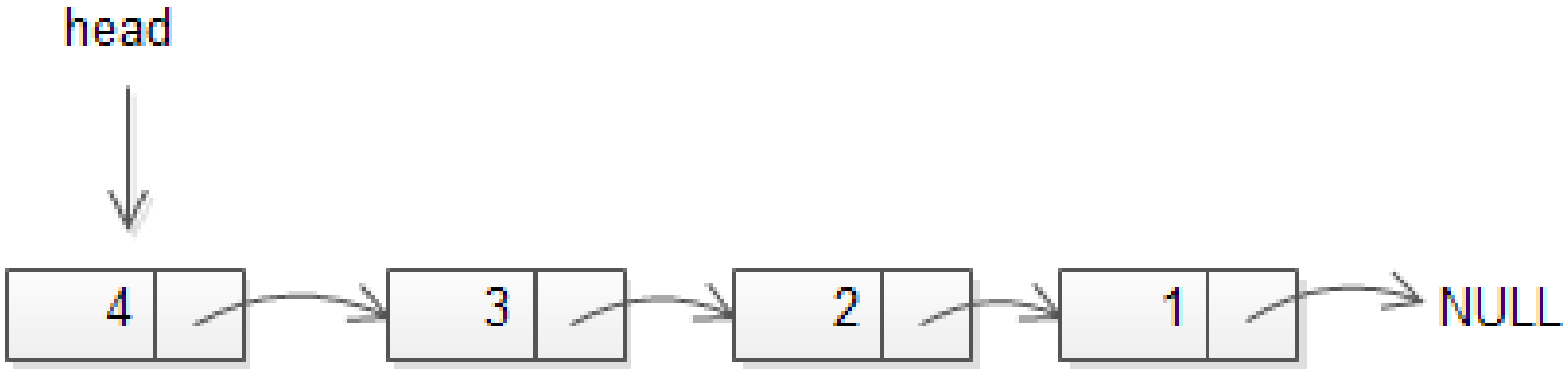
Adding a new node to a linked list (Prepend)




```
node * prepend(node * head, node * newnode)
{
    newnode->next = head;    // works even if the
list is empty
    return newnode;              // head changed !!
}
```

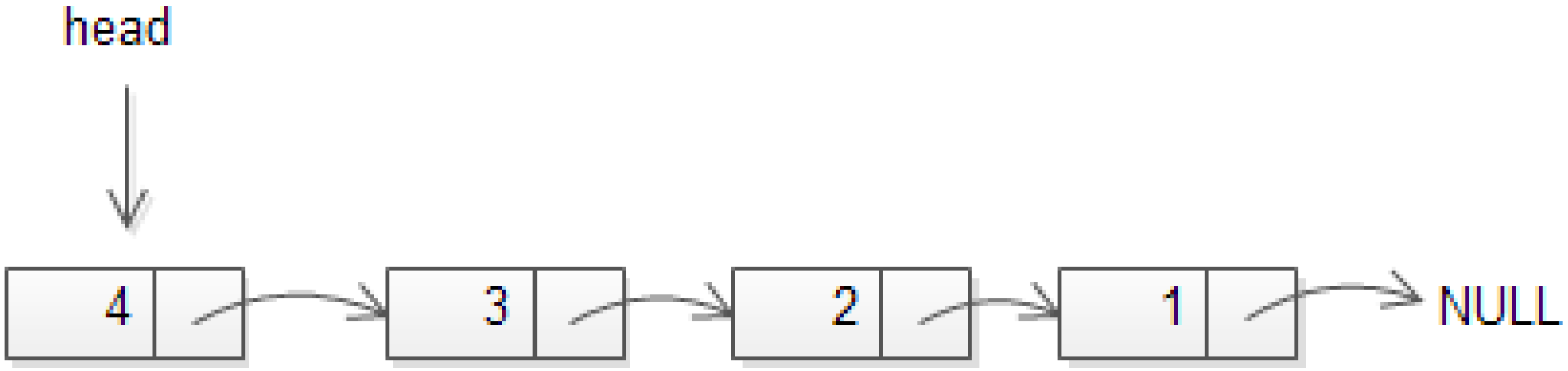
How to find the last element of a linked list?

```
node * find_last(node * head)
{
    // How?
}
```



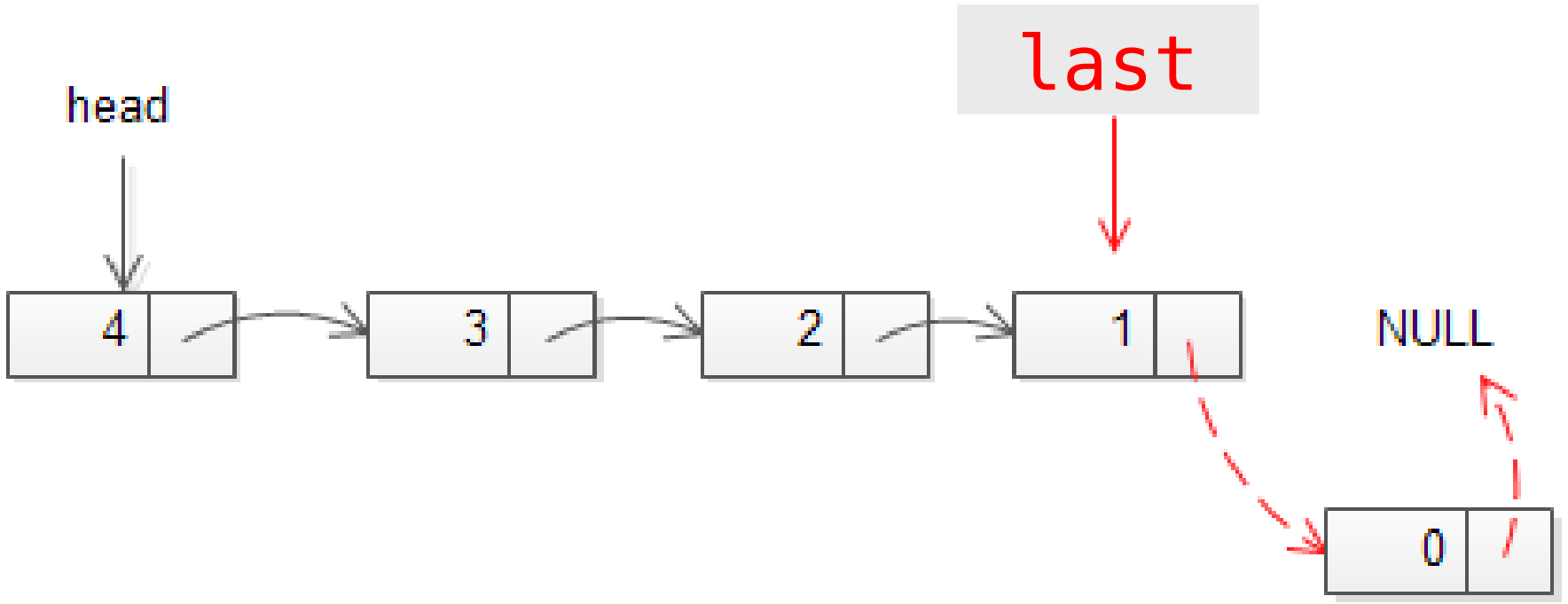
How to find the last element of a linked list?

```
node * find_last(node * head)
{
    if (head != NULL) {        // only if the list is not empty
        while (head->next != NULL)
            head = head->next;
    }
    return head;
}
```

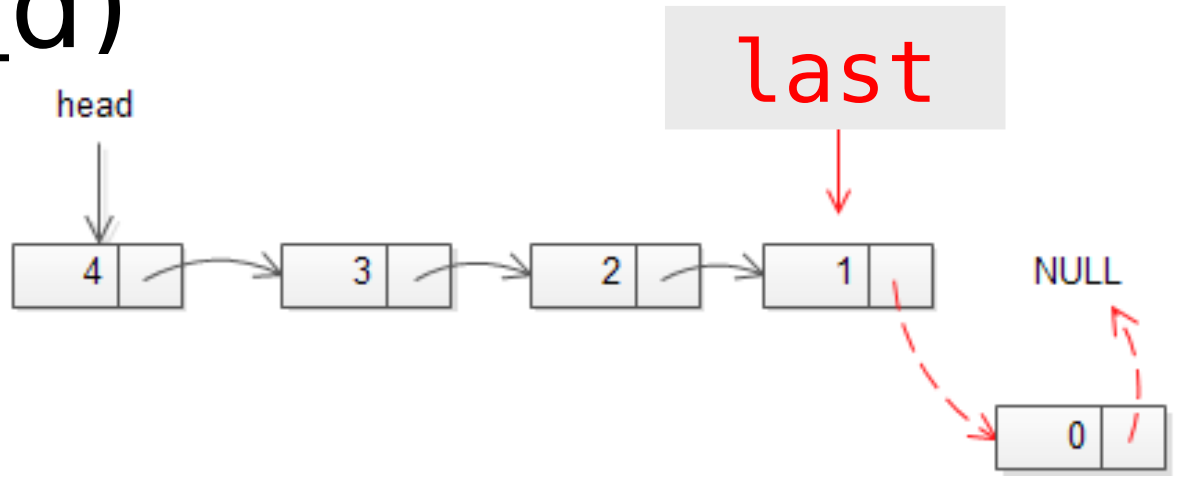


Adding a new node to a linked list (Append)

What if we want to add a new node to the end of list?



Adding a new node to a linked list (Append)




```
node * append(node * head, node * newnode) {
    node *last = find_last(head);       // find the last one
    if (last == NULL)       // if the list is empty, newnode is the head
        return newnode;
    last->next = newnode;
    newnode->next = NULL;
    return head;     // return the (unchanged) head
}
```

Arrays vs Linked Lists

Array	Linked List
Fixed Size, Resizing Expensive	Dynamic Size
Memory allocated in advance	Memory allocated dynamically
Sequential access fast	Sequential access slow (elements not in contiguous memory location)

2. Enumeration Types and Constants

Enumeration types

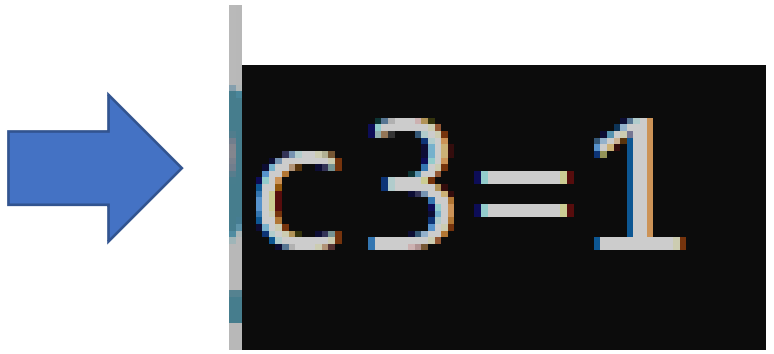
- The keyword **enum** is used to declare enumeration types.
- Treated as integers.
- By default the first one is 0, and each succeeding one has the next integer value.

- Names look like C identifiers
- User-defined integer-like types:

```
typedef enum {
  Red, Orange, Yellow, Green, Blue, Violet
} Color;
```

Enum Code Example

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

typedef enum {
    Red, Orange, Yellow, Green, Blue, Violet
} Color;


void main() {
    Color c1 = Red;
    Color c2 = Orange;
    int c3 = c1 + c2;
    printf("c3=%d\n", c3);
}
```

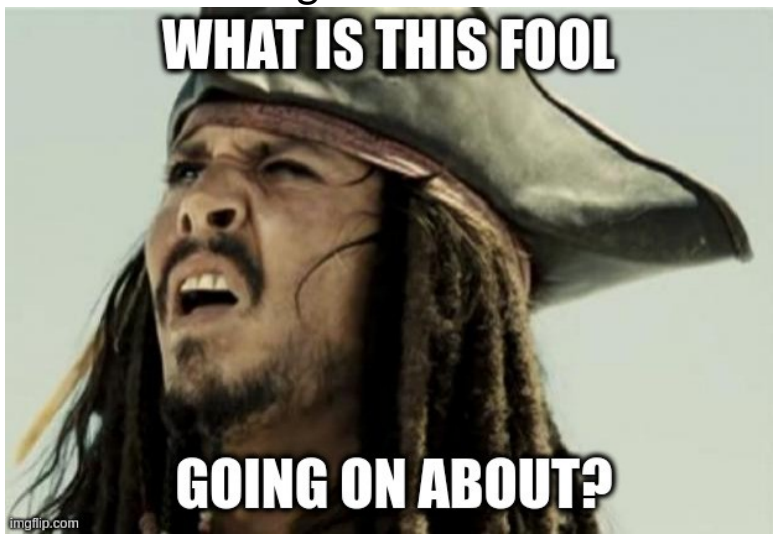


Technically has a value of 0

Technically has a value of 1

What is the point of enums?

Programmers first hearing about enums:



"The entire point of enums are to make your code more readable....instead of going back and forth checking what int values you may have, you can have a clear set of enums to make the code more readable."

- Enums = syntactic sugar

Source: https://stackoverflow.com/questions/8010492/what-is-the-point-of-enums

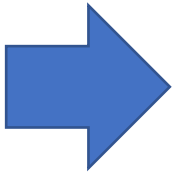
Type qualifier: const (1)

```
// constant int
const      int a = 10;          // cannot change a


// a pointer to a constant int
const int  *pa = &a;     // can change pa, but not
*pa
```

Constant Code Example 1

A pointer to a constant is NOT constant. So we can make pa point to a, then we can make pa point to b.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   void main() {
5       //define two constants
6       const    int    a = 10;
7       const    int    b = 7;
8       //pointer to a constant
9       const int* pa = &a;
10      printf("Val1=%d\n", *pa);
11      //pointer can point to other places
12      pa = &b;
13      printf("Val2=%d\n", *pa);
14  }
```



```
Val1=10
Val2=7
```

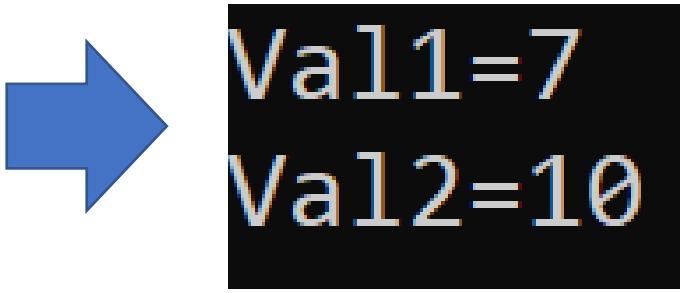
Type qualifier: const (2)

```
// a constant pointer to an int
int  * const pb = &b;    // can change *pb, but not pb


// a constant pointer to a constant int
const int  * const pc = &a; // cannot change *pc or
pc
```

Constant Code Example 2

A constant pointer is CONSTANT and cannot be changed.
i.e. it always points to one fixed address in memory
However, we can change what is stored AT that address.

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

void main() {
    //define two integers
    int b = 7;
    //constant pointer, cannot be changed
    int* const pb = &b;
    printf("Val1=%d\n", *pb);
    //however the value AT the address can be changed
    b = 10;
    printf("Val2=%d\n", *pb);
}
```



3. Function Pointers

Least complicated pointer program in C:

Function Pointers

```
/* function returning integer */
int func();

/* function returning pointer to integer */
int * func();

/* pointer to function returning integer */
int (*func)();

/* pointer to function returning pointer to int */
int * (*func)();
```

Example of using a pointer to a function

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

int mymax(int a, int  b)
{
    if (a > b) {
        return a;
    }
    else {
        return b;
    }
}

void main() {
    // a pointer to function
    int (*pf)(int a, int b);
    // assign a value to the pointer
    pf = mymax;       // C99 style. Note that it is NOT mymax()
    int val = pf(3, 5);
    printf("Val=%d\n", val);
}
```

Alternative Syntax for using a pointer to a function

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

int mymax(int a, int  b)
{
    if (a > b) {
        return a;
    }
    else {
        return b;
    }
}

void main() {
    // a pointer to function
    int (*pf)(int a, int b);
    // assign a value to the pointer
    pf = mymax;      // C99 style. Note that it is NOT mymax()
    int val = (*pf)(3, 5);
    printf("Val=%d\n", val);
}
```

Why do we care about such complicated behavior?

Example: quicksort in C library

- The prototype (in <stdlib.h>)

```
void qsort(void * base,
           size_t nel,
           size_t width,
           int (*compare)(const void *, const void *));
```

- qsort takes…
  - base: the address of the array as an untyped pointer
  - nel: the number of elements in the array
  - width: the size (in byte) of ONE element of the array
  - compare: a **pointer to a function** that compares two values

Why is having a pointer to a function good?

Why is having a pointer to a function good?

```
void qsort(void * base,
           size_t nel,
           size_t width,
           int (*compare)(const void *, const void *));
```

qsort() only knows it is asked to sort nel items, each having width bytes.
It does not know the type of elements or how to compare them.

- Need to tell qsort() how to compare items in the array
  - **We have a *generic* quickSort implementation**
    - Do not want to implement one for each type of data
- The qsort() implementation calls the comparator to rank elements

```
int (*compare)(const void *a, const void *b);
```

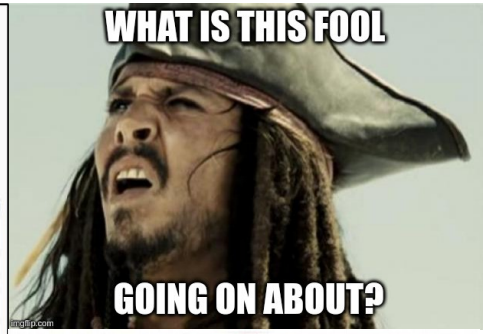
  - The function takes the address of two items to be compared,
  - and returns:
    - 0 if *a EQUALS *b
    - A positive value if *a is GREATER THAN *b
    - A negative value if *a is LESS THAN *b

Example of the Compare Function

When qsort() needs to compares two items, it provides their addresses

```
int compare_int(const void *a, const void *b)
{ // qsort() does not know the type, but you know
     return *(int *)a - *(int *)b;
}

int compare_double(const void *a, const void *b)
{ // qsort() does not the type, but you know
   double va, vb;
   va = *(double *)a; vb = *(double *)b;
     return va > vb ? 1 : (va < vb ? -1 : 0);
}
```

Lecture Conclusions



- Conclusion 1: You should understand the linked list and how using a pointer to denote the head makes manipulation of the nodes straight forward.
- Conclusion 2: Understand how to use enumerators and constants.
- Conclusion 3: Start to understand how we can pass functions as pointers for flexible coding.

Figure Sources

- Figures for linked list are from http://www.zentut.com/c-tutorial/c-linked-list/

1. https://programmercave0.github.io/assets/Memes-Linkedlist/llmeme2.jpg
2. https://programmercave0.github.io/assets/Memes-Linkedlist/llmeme6.jpg
3. https://akashicseer.com/wp-content/uploads/2022/02/pirate-what-is-this-fool.jpg
4. https://www.idlememe.com/wp-content/uploads/2021/10/spiderman-pointing-meme-idlememe-2.jpg