# DYNAMIC MEMORY ALLOCATION
# LINKED LISTS

Problem Solving with Computers-I

https://ucsb-cs16-wi17.github.io/

```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

GitHub

# Program layout in memory at runtime

A generic model for memory

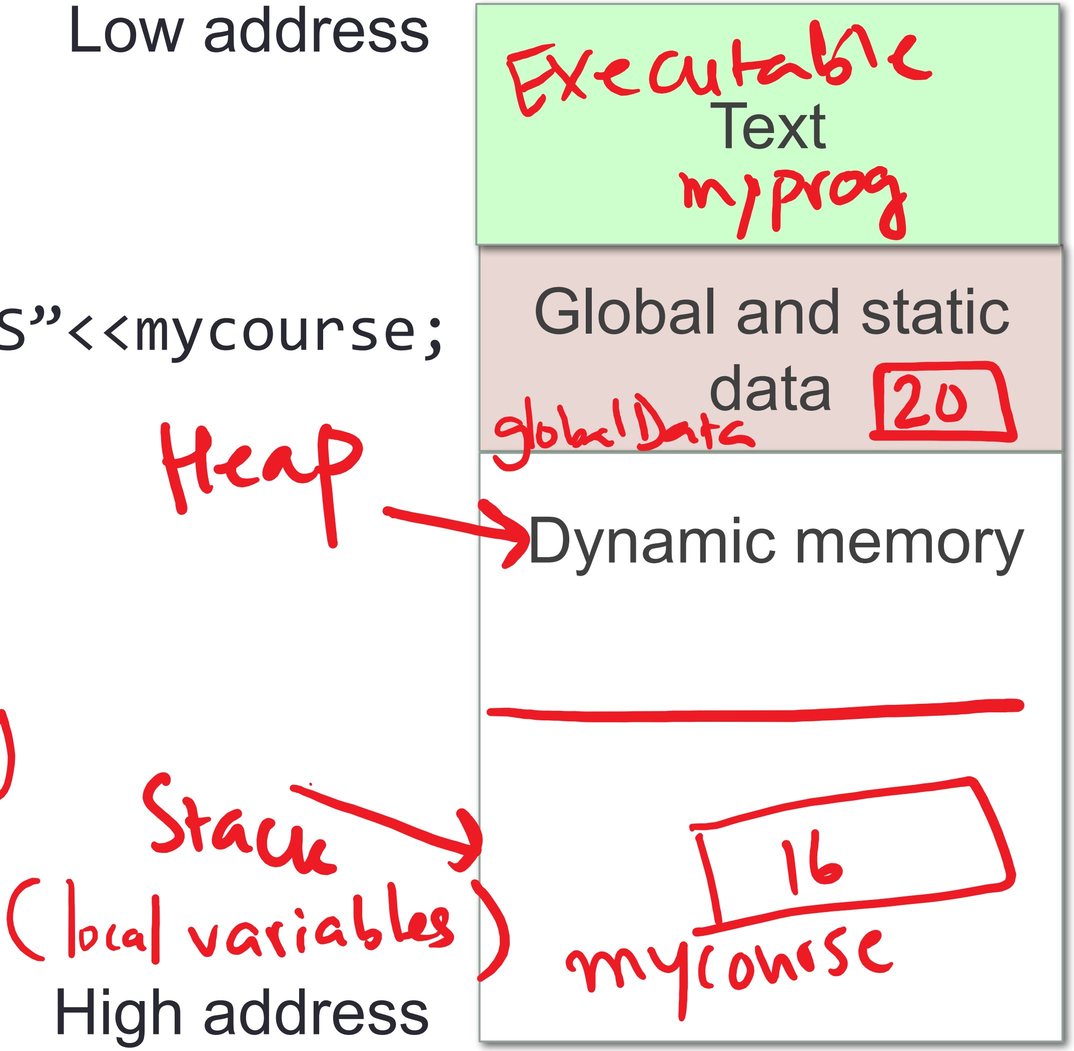
0
1
2
3
4
5
6
7
8
9
10

```
int globalData=20;
int foo() {
   int mycourse = 16;
   cout<<“Welcome to CS”<<mycourse;
}
```

|| compile

./myprog

Data on heap is explicitly allocated and deallocated by the programmer



Low address

Executable

Text

myprog

Global and static data

globalData

20

Heap

Dynamic memory

Stack (local variables)

16

mycourse

High address

# Creating data on the heap: new and delete

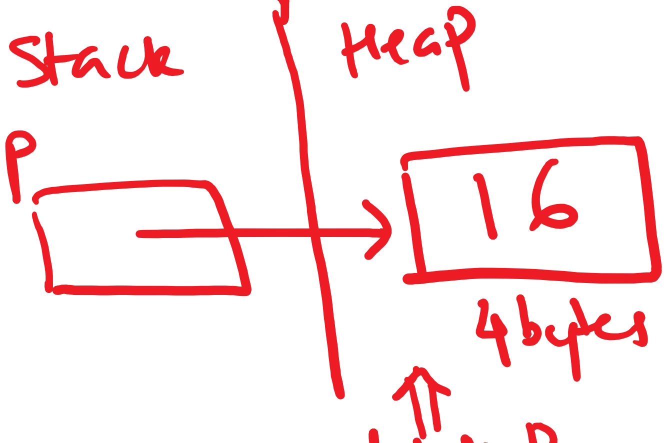
```
int foo() {
   int mycourse = 16;
   cout<<“Welcome to CS”<<mycourse;
}
```

To allocate memory use: 'new'

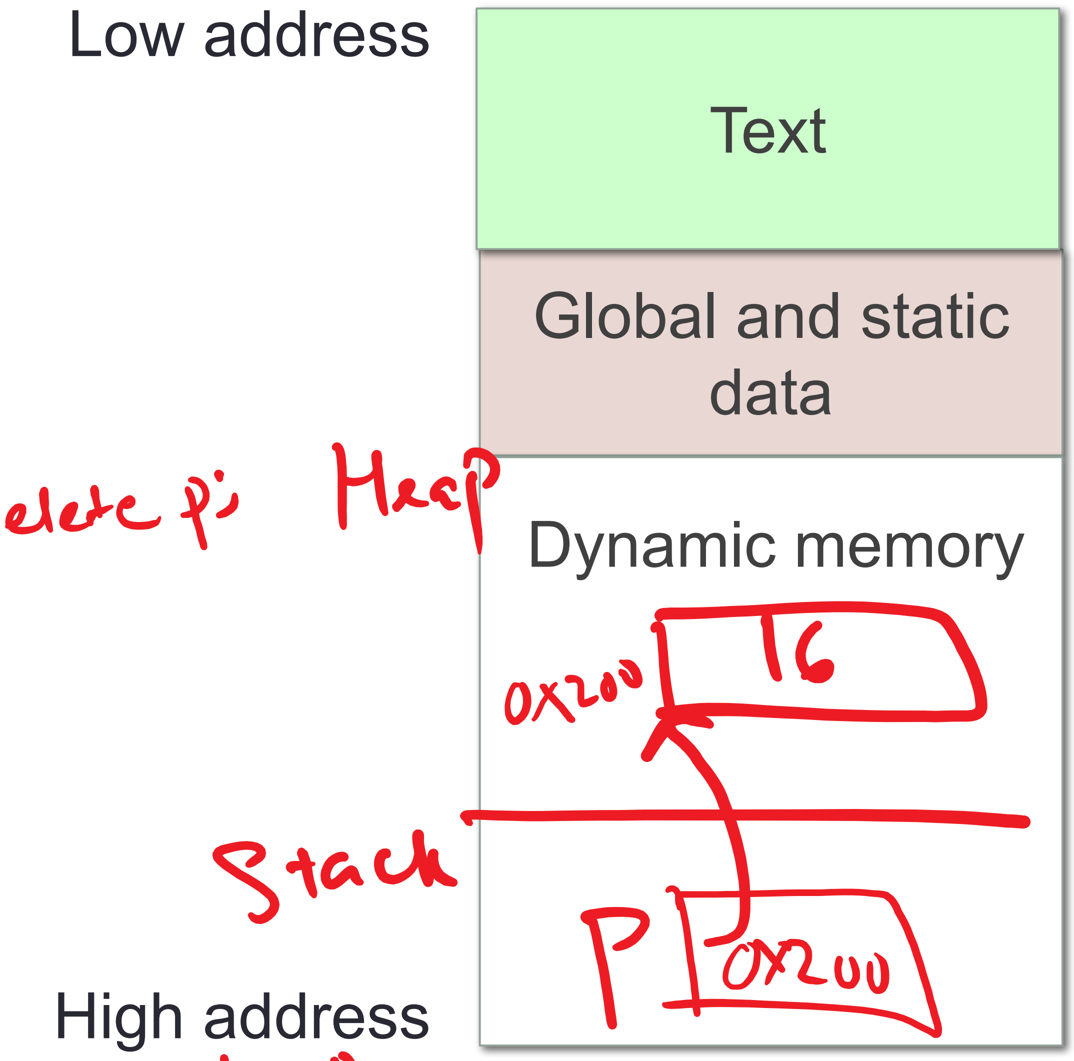
To deallocate use : 'delete'

```
int foo() {
  int *p = new int;
   *p = 16;
   cout<<*p;
  delete p; //
}
```

State of memory before delete p;



delete p
deallocates memory on heap

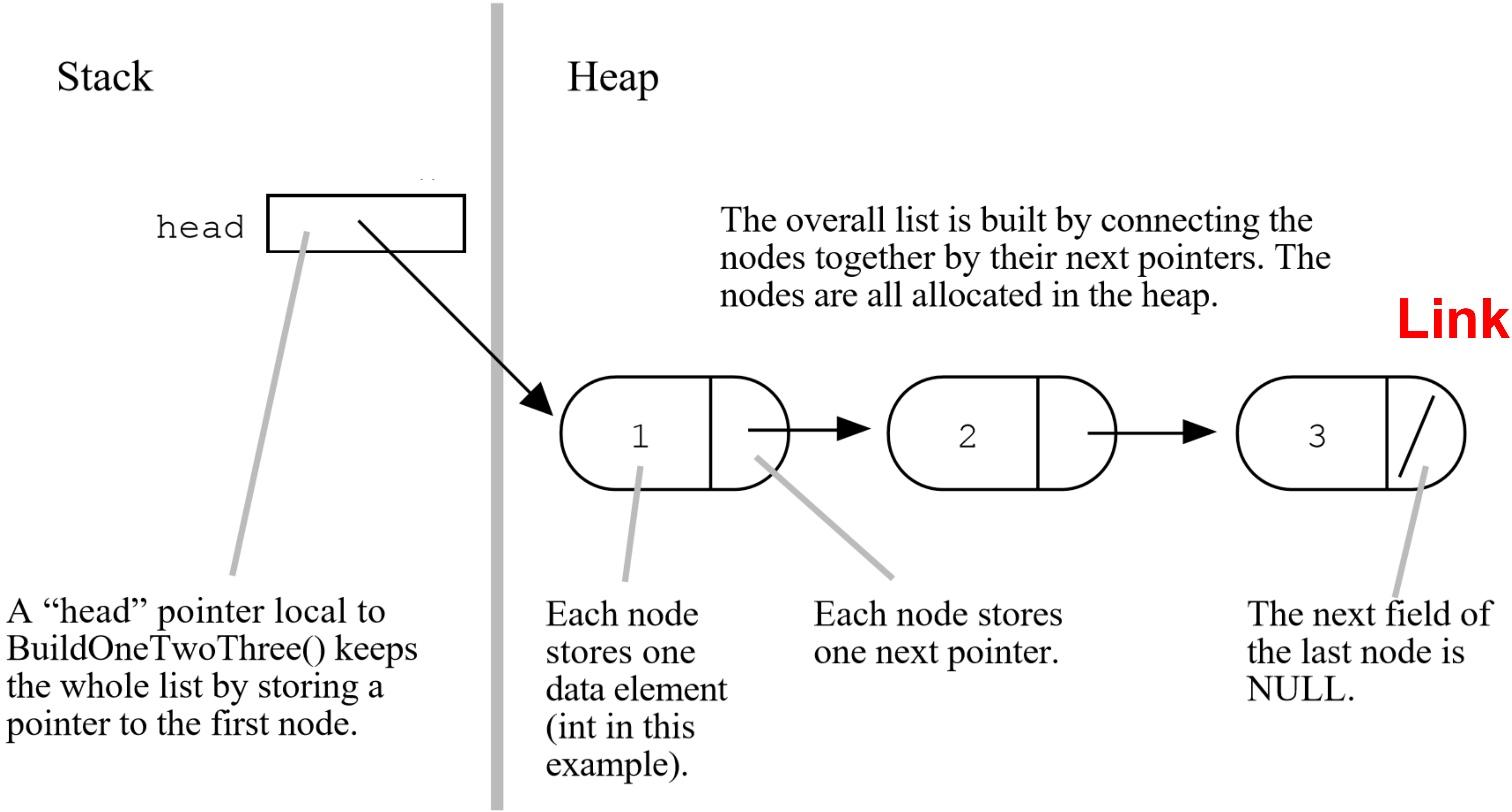


# Linked Lists

**The Drawing Of List {1, 2, 3}**

| | | |
|---|---|---|
| 1 | 2 | 3 |

**Array List**



The overall list is built by connecting the nodes together by their next pointers. The nodes are all allocated in the heap.

**Linked List**

A “head” pointer local to BuildOneTwoThree() keeps the whole list by storing a pointer to the first node.

Each node stores one data element (int in this example).

Each node stores one next pointer.

The next field of the last node is NULL.

# Creating nodes (Stack vs. Heap)

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

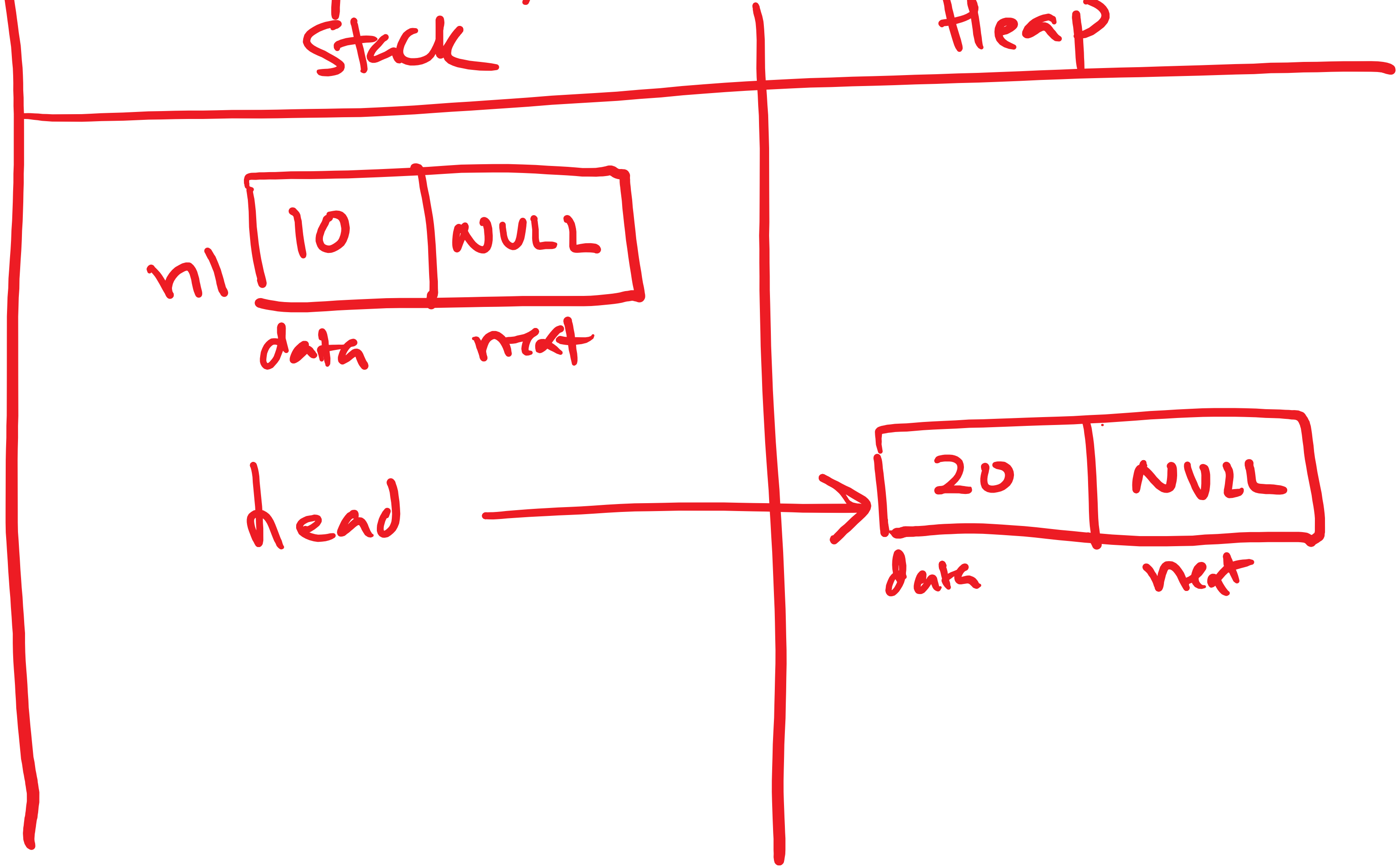
Code

```
Node n1;

n1.data = 10;
n1.next = NULL;

Node *head = new Node;
head->data = 20;
head->next = NULL;
```

Memory diagram



# Building a list from an array

a

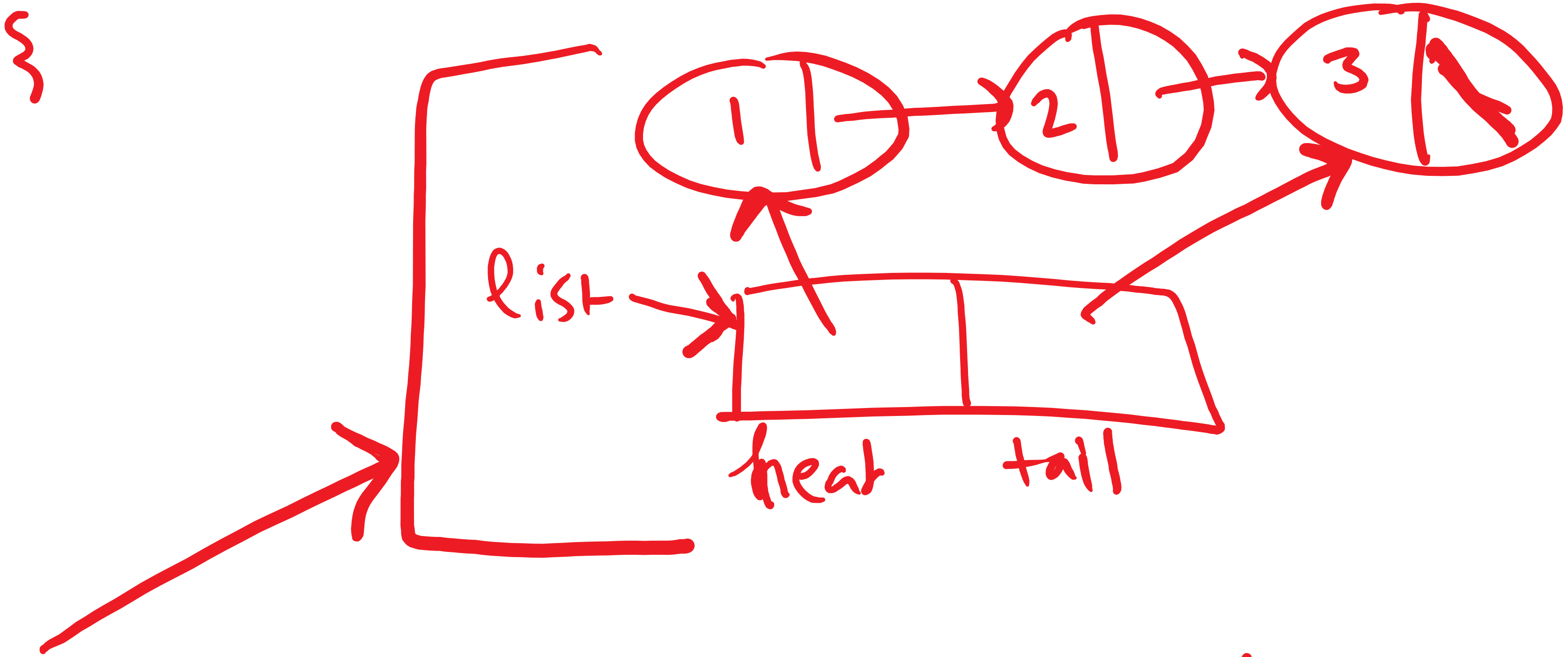
| | | |
|---|---|---|
| 1 | 2 | 3 |

size

3

```
LinkedList * arrayToLinkedList(int a[], int size) ;
```

```
struct LinkedList {
        Node* head;
        Node * tail;
};
```

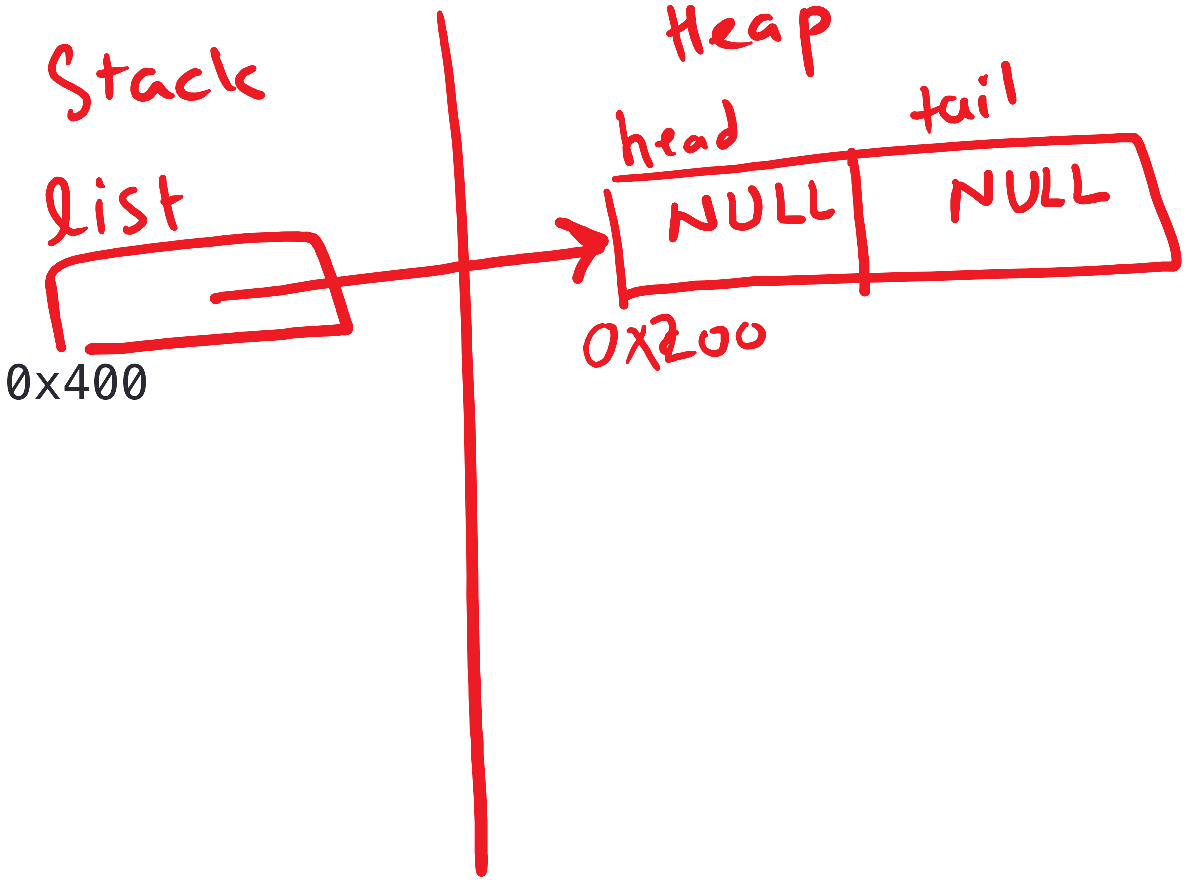


Desired output for given input

# Building a list from an array

From live coding session

```
LinkedList * arrayToLinkedList(int *a, int size) {// Empty list case
    LinkedList *list = new LinkedList;
    list->head = list->tail = NULL;
    return list;
}
```



0x400

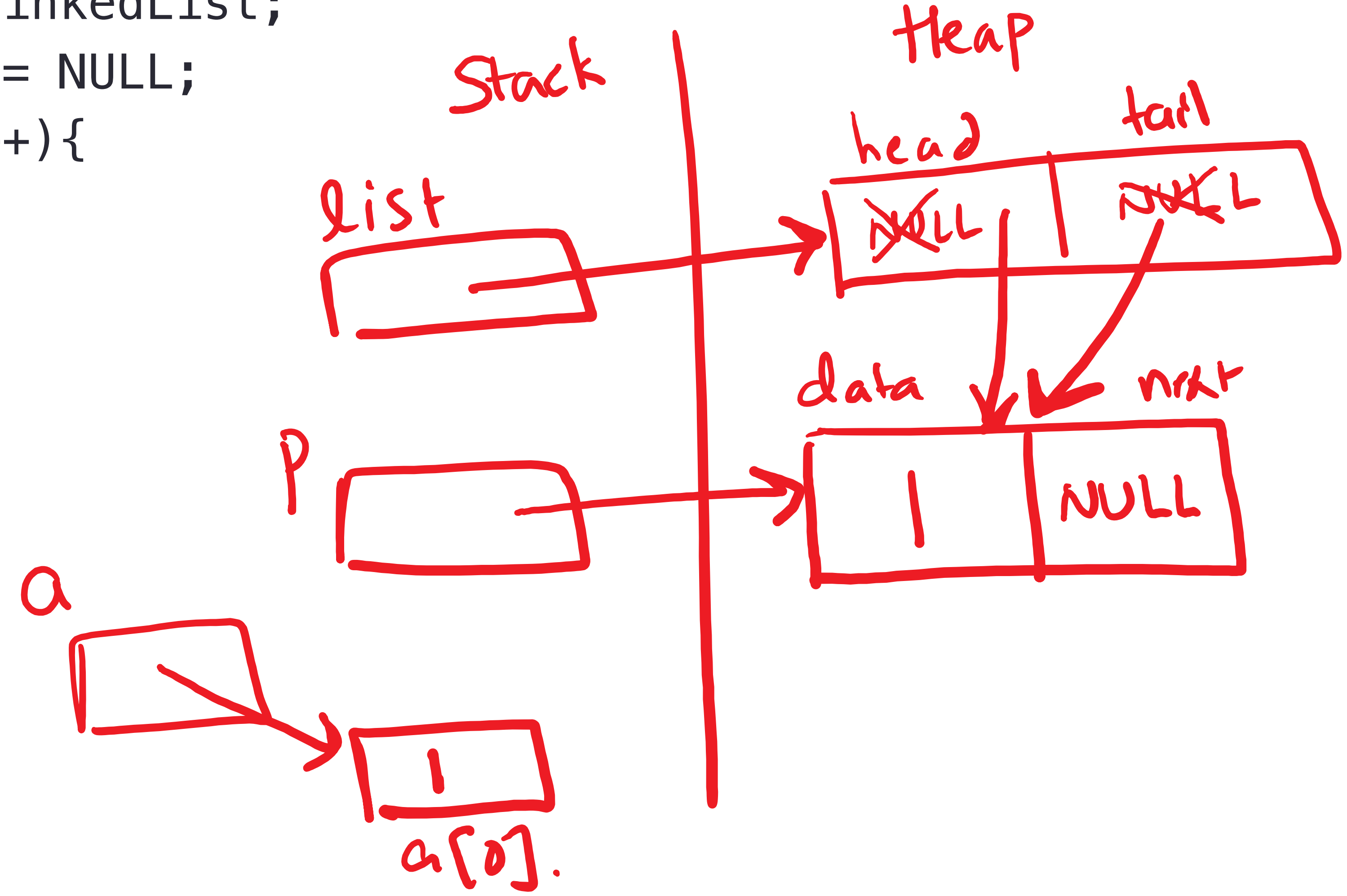
What is the return value of the function?

(See diagram on the right)

[✓ A. 0x200]

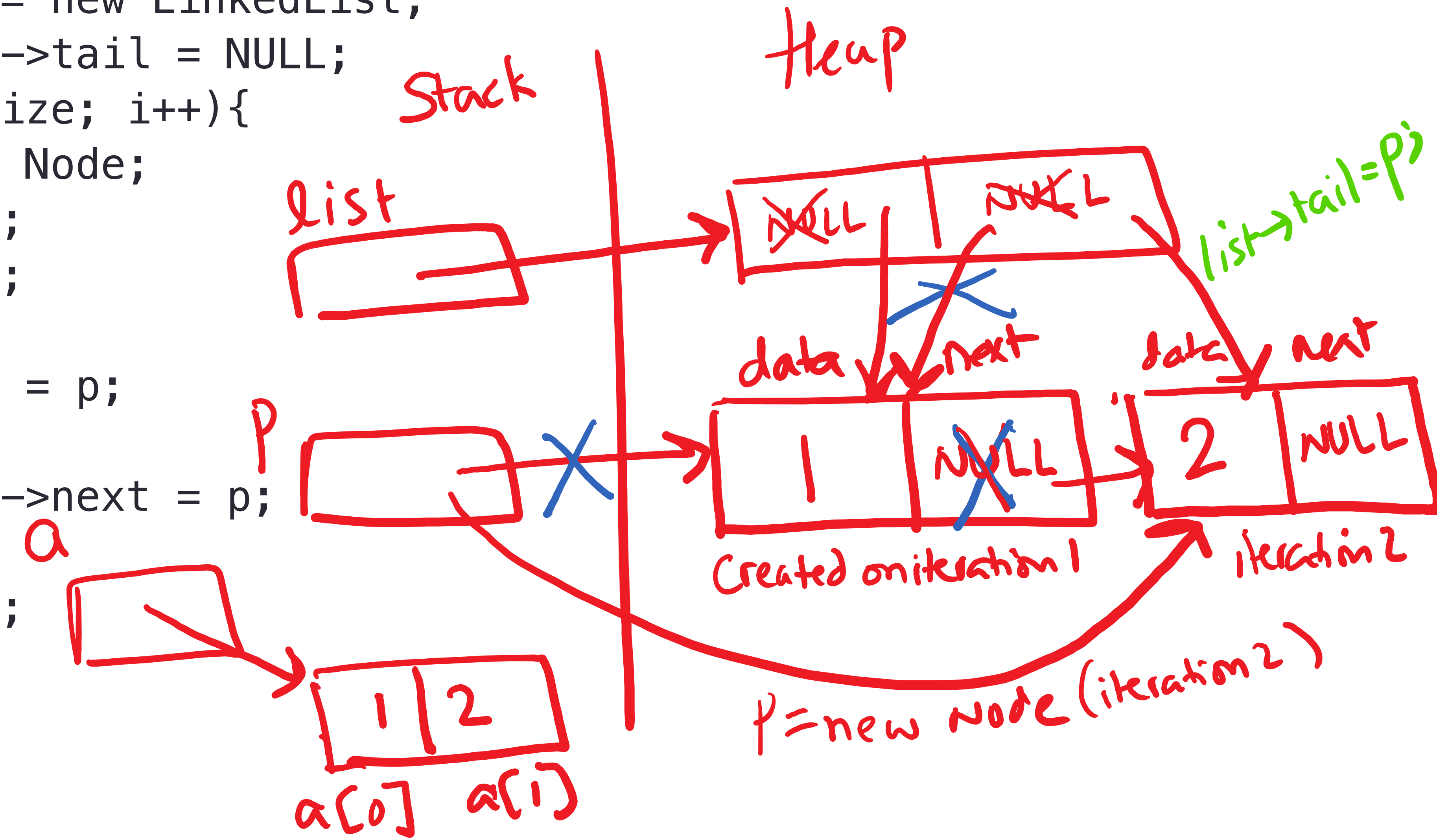
B. 0x400

```
LinkedList * arrayToLinkedList(int *a, int size){
    LinkedList *list = new LinkedList;
    list->head = list->tail = NULL;
    for(int i=0; i< size; i++){
       Node* p  = new Node;
       p->data = a[i];
       p->next = NULL;
       list->head = p;
       list->tail = p;
    }
    return list;
}
```



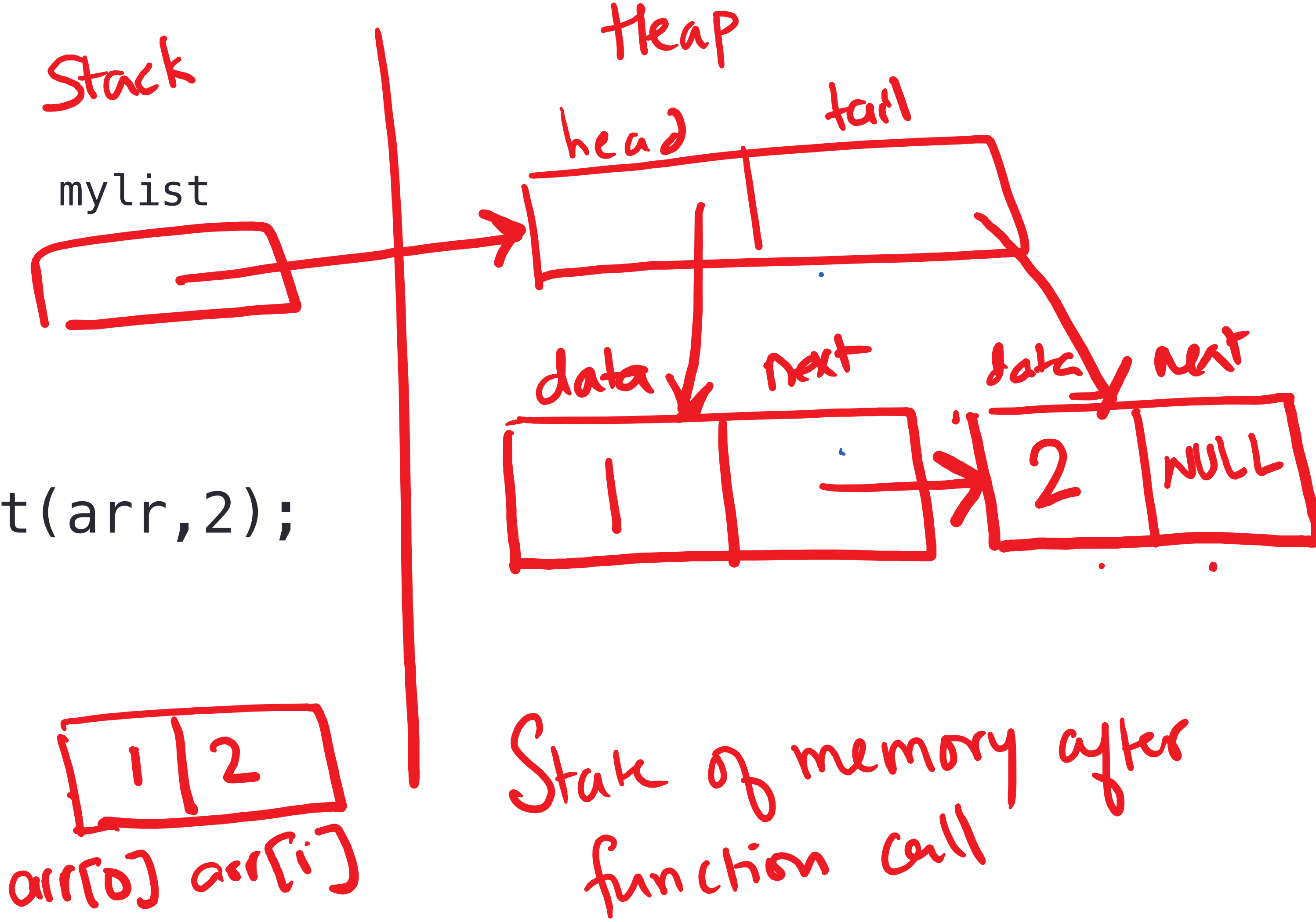
// Input → array of size 1

```
LinkedList * arrayToLinkedList(int *a, int size) {
    LinkedList *list = new LinkedList;
    list->head = list->tail = NULL;
    for(int i=0; i< size; i++){
        Node* p  = new Node;
        p->data = a[i];
        p->next = NULL;
        if(i==0){
            list->head = p;
        }else{
            list->tail->next = p;
        }
        list->tail = p;
    }
    return list;
}
```



```
LinkedList * arrayToLinkedList(int *a, int size) ;
```

```
int arr[2] ={1, 2};
LinkedList *mylist;
mylist = arrayToLinkedList(arr,2);
```

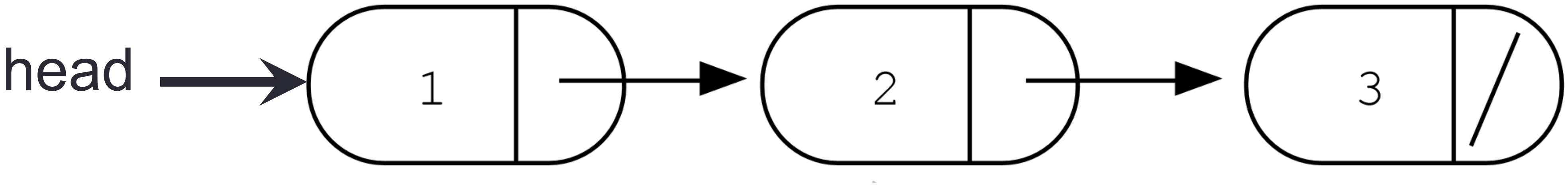


State of memory after function call

# Accessing elements of a list

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

Node * head ;



Assume the linked list has already been created, what do the following expressions evaluate to?

1. head->data 1
2. head->next->data 2
3. head->next->next->data 3
4. head->next->next->next->data (Derefencing a NULL pointer)

A. 1
B. 2
C. 3
D. NULL
[✓ E. Run time error]

# Next time

- More on linked lists
- Dynamic arrays
- Pointer arithmetic
- Dynamic memory pitfall