

# UESTC1005 - Introductory Programming

Structures, Bit Manipulation and  
Linked Lists

Lecture 13

*Dr Ahmed Zoha*

*Lecturer in Communication Systems*

*School of Engineering*

*University of Glasgow*

*Email: [ahmed.zoha@Glasgow.ac.uk](mailto:ahmed.zoha@Glasgow.ac.uk)*

# (RECAP) Accessing members in Structure

- A structure variable name is not a pointer
  - You need special syntax to access the members
- You access the member of the structure by **writing the variable name followed by a period**, followed by the member variable name
  - The period is called the member selection operator (aka dot operator)
  - There are no spaces permitted between the variable name, the period, and the member name.
  - To set the value of title in book 1, you write
    - `book1.title = “Spring”;`
    - `book1.author = “ahmed”;`

# (RECAP) Display Date Struct Example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      struct date
7      {
8          int month;
9          int day;
10         int year;
11     };
12
13     struct date today;
14
15     today.month = 12;
16     today.day = 03;
17     today.year = 2019;
18
19     printf("Today is date: %i-%i-%i\n", today.month, today.day, today.year);
20 }
21
```

Today is date: 12-3-2019

Process returned 0 (0x0) execution time : 0.193 s  
Press any key to continue.

# (RECAP) Initializing Structures

- Initializing structures is similar to initializing arrays
  - The elements are listed inside a pair of braces, with each element separated by comma
  - E.g.; `struct date today = {7,2,2019};`
- Just like an array initialization, fewer values might be initialized than are contained in the structure
  - Example: `struct date date1 = {12, 10};` which sets the `date1.month` to 12, `date1.date` to 10, and no value to `date1.year`.
- Another method is to use `.member = value` method
  - Example `struct date dates1 = {.month =12, .date=10};`


# Nested Structures

- Nested structures are simply *structures inside of structure*
- We have already seen how to logically group month, date and year into a structure called **date**
  - What if we want to group time into hours, minutes and seconds

struct time

```
{  
    int hours;  
    int minutes;  
    int seconds;  
};
```

Can we group date and time  
Together as nested structure



```
struct datetime  
{  
    struct date sdate;  
    struct time stime;  
};
```

The variables can now be defined as struct  
dateandtime

E.g. : struct datetime event;  
We can access the elements simply by  
event.sdate;  
event.sdate.month = 10;  
++event.stime.seconds;

## Initialization in nested structure

```
struct datetime event = {{2,1,2015},{3,30,0}};  
OR when order does not matter  
struct datetime event  
{ {.month =2, . date=1, .year=2015},  
  {.hours = 3, . Minutes= 30, .seconds=0}  
};
```

## Declaration of structure within a structure

```
struct Time  
{  
    struct date  
    {  
        int month;  
        int days;  
        int year;  
    } dob;  
  
    int hours;  
    int minutes;  
    int seconds;  
};
```

# Structures as Pointers

- C allows for pointers to structures
- You can define a variable to be pointer to structure as
  - `struct date todaysDate, *datePtr;`
  - We can simply set the `datePtr` to point to `todaysDate` using assignment operator : `datePtr = &todaysDate;`
  - The members of the date structure can be assessed by the pointer `datePtr` using **arrow (->) operator**.
  - **Example:** `datePtr -> month = 12;` is equal to `(*datePtr).month = 12;`
  - **Parentheses are important because dot takes precedence, therefore you will always see an arrow operator with pointers to structures in the code.**

# Example: pointers to structure

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct date
5  {
6      int month;
7      int date;
8      int year;
9  };
10
11 int main()
12 {
13     struct date today, *datePtr;
14
15     datePtr = &today;
16
17     datePtr->month = 12; // Setting the values which is
18     datePtr->date=12; // equivalent of dereferencing a pointer
19     datePtr->year=2019; // using (*datePtr)
20
21     //Displaying the set date
22
23     printf("Today's date is %i/%i/%i\n", datePtr->date, datePtr->month, datePtr->year);
24 }
25
```

Today's date is 12/12/2019

Process returned 0 (0x0) execution time : 0.229 s  
Press any key to continue.



# structure to functions as arguments

- We can pass structure as an argument to the function

```
struct family
{
    char name[20];
    int age;
    char father[20];
    char mother[20];
};

bool siblings (struct family member1, struct family member2)
{
    if (strcmp(member1.mother, member2.mother)==0)
        return true;
    else
        return false;
}
```

Drawbacks: It can take quite a bit of time copying large structures as arguments, As well as this is not memory efficient

# Pointers to structure as function arguments

- Pointers to structures as function arguments minimize the memory consumption and copying time.
  - All you have to do is copy the address

```
bool siblings (struct family *pmember1, struct family *pmember2)
{
    if (strcmp(pmember1->mother, pmember2->mother)==0)
        return true;
    else
        return false;
}
```

# Reminder

- It is recommended to use pointers while passing structures to a function
  - Old implementation of C do not support passing of structures directly
- The downside is that you have less protection of data
  - But you can always use const qualifier to solve the problem
- Advantages of passing structure as an argument
  - Functions works with the copies of original data which is safer
  - Program is more readable
- Main disadvantages are
  - Older implementation might not handle the code
  - Waste time and space
  - Especially wasteful to pass large structures when you are only using one or two members

# Examples: Passing Pointers to structure in Function arguments

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct item
5  {
6      char *itemName;
7      int qty;
8      float price;
9      float amount;
10 };
11
12 void readItem(struct item *i);
13 void printItem(struct item *i);
14
15 int main()
16 {
17     struct item itm;
18     struct item *pItem;
19
20     pItem = &itm;
21
22     pItem->itemName = (char *) malloc(30 * sizeof(char));
23
24     if(pItem == NULL)
25         exit(-1);
26
27     // read item
28     readItem(pItem);
29
30     // print item
31     printItem(pItem);
32
33     free(pItem->itemName);
34
35     return 0;
36 }
```

# Examples: Passing Pointers to structure in Function arguments

```
36
37
38 void readItem(struct item *i)
39 {
40     printf("Enter product name: ");
41     scanf("%s", i->itemName);
42
43     printf("\nEnter price: ");
44     scanf("%f", &i->price);
45
46     printf("\nEnter quantity: ");
47     scanf("%d", &i->qty);
48
49     i->amount = (float)i->qty * i->price;
50 }
51
52 void printItem(struct item *i)
53 {
54     /*print item details*/
55     printf("\nName: %s", i->itemName);
56     printf("\nPrice: %.2f", i->price);
57     printf("\nQuantity: %d", i->qty);
58     printf("\nTotal Amount: %.2f", i->amount);
59 }
60
```

```
Enter product name: Apple
Enter price: 10000
Enter quantity: 3
Name: Apple
Price: 10000.00
Quantity: 3
Total Amount: 30000.00
Process returned 0 (0x0)   execution time : 16.593 s
Press any key to continue.
```

# bitwise operations/ bit Manipulation

# Bitwise Operators

- The **& (bitwise AND)** : Takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
- The **| (bitwise OR)**: Takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
- The **^ (bitwise XOR)**: Takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
- The **<< (left shift)**: Takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
- The **>> (right shift)**: Takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
- The **~ (bitwise NOT)** : Takes one number and inverts all bits of it

# Truth Table for $\&$ | $\wedge$

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Bit 1	Bit 2	Bit 1   Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Bit 1	Bit 2	Bit 1 $\wedge$ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0



# Basic Example of bitwise operations

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      // a = 5(00000101), b = 9(00001001)
6      unsigned char a = 5, b = 9;
7      // The result is 00000001
8      printf("a = %d, b = %d\n", a, b);
9      printf("a&b = %d\n", a & b);
10     // The result is 00001101
11     printf("a|b = %d\n", a | b);
12     // The result is 00001100
13     printf("a^b = %d\n", a ^ b);
14     // The result is 11111010
15     printf("~a = %d\n", a = ~a);
16     // The result is 00010010
17     printf("b<<1 = %d\n", b << 1);
18     // The result is 00000100
19     printf("b>>1 = %d\n", b >> 1);
20
21     return 0;
22 }
23
```

```
= 5, b = 9
&b = 1
|b = 13
^b = 12
a = 250
<<1 = 18
>>1 = 4
```

```
process returned 0 (0x0)  execution time : 0.213 s
press any key to continue.
```

# Linked List Definition and its Advantages

- A linked list is a dynamic data structure where each element (called a node) is made up of two items - the data and a reference (or pointer) which points to the next node. A linked list is a collection of nodes where each node is connected to the next node through a pointer. The first node is called a head and if the list is empty then the value of head is NULL.

## **Some of the advantages of using a linked list are**

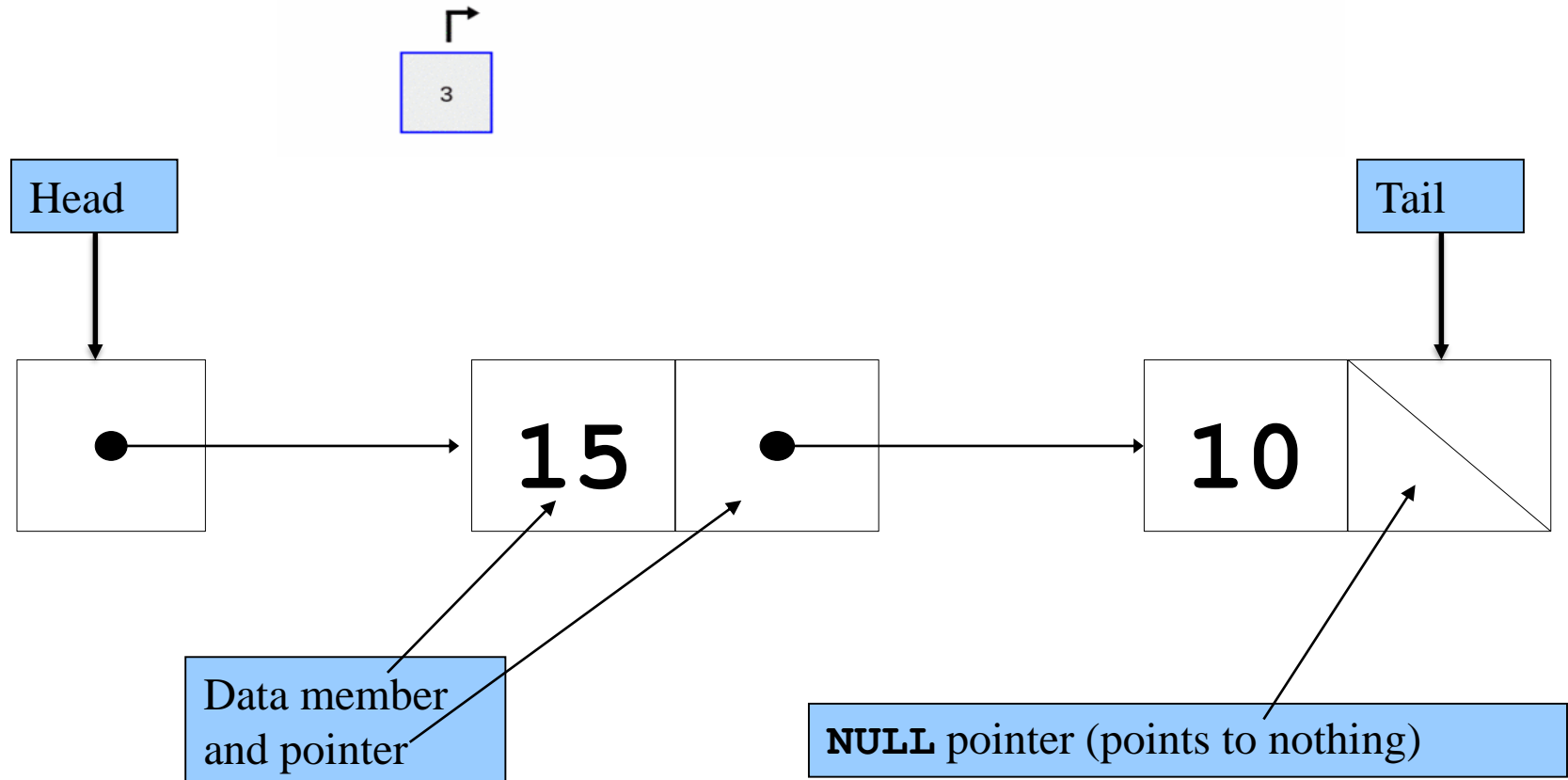
- In contrast to arrays, which have pre-defined or fixed length, linked lists have a dynamic length which can be increased or decreased at runtime.
- Insertion and deletion operations in the linked list are much faster in comparison to other data structure such as the queue, stack, and arrays.

# Linked List Representations

- The first node is called head and if the linked list is empty then the value of head is NULL.
- Each node consists of
  - Data
  - Pointer to next node
- We represent nodes using structures as shown below

```
// A linked list node
struct Node {
    int data;
    struct Node* next;
};
```

# Singly Connected Linked List



# Linked List Representation in C (1/3)

```
int main()
{
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // allocate 3 nodes in the heap
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    /* Three blocks have been allocated dynamically.
       We have pointers to these three blocks as head,
       second and third
```

head	second	third
+---+-----+	+---+-----+	+---+-----+
#   #	#   #	#   #
+---+-----+	+---+-----+	+---+-----+

# Linked List Representation in C (2/3)

```
# represents any random value.  
Data is random because we haven't assigned  
anything yet */  
  
head->data = 1; // assign data in first node  
head->next = second; // Link first node with  
// the second node  
  
/* data has been assigned to the data part of the first  
block (block pointed by the head). And next  
pointer of first block points to second.  
So they both are linked.
```

head		second		third
+---+---+		+---+---+		+---+---+
1   o----->		#   #		#   #
+---+---+		+---+---+		+---+---+

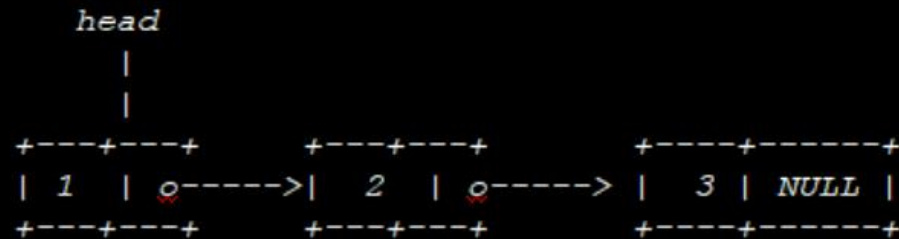
```
*/  
// assign data to second node  
second->data = 2;  
  
// Link second node with the third node  
second->next = third;
```

# Linked List Representation in C (3/3)

```
third->data = 3; // assign data to third node
third->next = NULL;
```

*/\* data has been assigned to data part of third block (block pointed by third). And next pointer of the third block is made NULL to indicate that the linked list is terminated here.*

*We have the linked list ready.*



*Note that only head is sufficient to represent the whole list. We can traverse the complete list by following next pointers. \*/*

```
return 0;
```

# Linked List Traversal

```
// This function prints contents of linked list starting from  
// the given node  
void printList(struct Node* n)  
{  
    while (n != NULL) {  
        printf(" %d ", n->data);  
        n = n->next;  
    }  
}
```

**Output: 1 2 3**

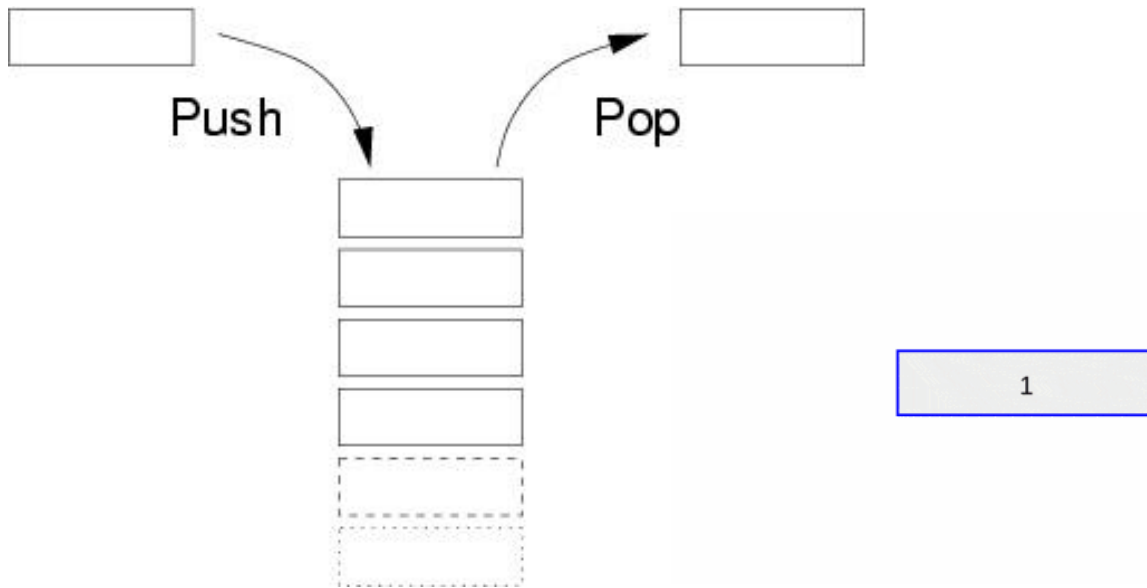


# Types of Linked Lists

- **Singly linked list**
  - Begins with a pointer to the first node
  - Terminates with a null pointer
  - Only traversed in one direction
- **Circular, singly linked**
  - Pointer in the last node points back to the first node
- **Doubly linked list**
  - Two “start pointers” – first element and last element
  - Each node has a forward pointer and a backward pointer
  - Allows traversals both forwards and backwards
- **Circular, doubly linked list**
  - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

# Definition of Stack

A stack is a data structure similar to real-world stacks (such as a pile of plates, books or a deck of cards). It is only possible to read a single element at a given time. It works according to the “**First In Last Out**” (FILO) mechanism. In this mechanism, the first inserted element is the last element to remove from the stack. The last inserted element is the first element to remove from the stack. It is also called “**Last In First Out**” (LIFO).



# Difference between Stack and Linked Lists

## STACK VERSUS LINKED LIST

STACK	LINKED LIST
An abstract data type that serves as a collection of elements with two principal operations which are push and pop	A linear collection of data elements whose order is not given by their location in memory
Push, pop and peek are the main operations performed on a stack	Insert, delete and traversing are the main operations performed on a linked list
Can read the topmost element	It is required to traverse through each element to access a specific element
First In last out	Elements connect to each other by references
Simpler than linked list	More complex than stack
	Visit <a href="http://www.PEDIAA.com">www.PEDIAA.com</a>

.“Source: ” By The original uploader was R. Koot at English Wikipedia. – Transferred from en.wikipedia to Commons ([CC BY-SA 3.0](https://creativecommons.org/licenses/by-sa/3.0/)) via [Commons Wikimedia](https://commons.wikimedia.org/wiki/File:Stack_and_Linked_Lists.png)

# Final Exam : Important Topics

- See on the board , and you must prepare well these topics for the Final Exam
- This last lecture is very important and I urge you to pay attention to the topics covered.
- Final Lab Report: You need to combine all your labs from 1 to 6 as **one pdf file** and upload it under file lab report submission.