

# Linked Lists: A dynamic data structure

---

Lecture 7

# Array versus Linked Lists

- ✉ Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - **Dynamic:** a linked list can easily grow and shrink in size.
    - ↳ We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - ↳ In contrast, the size of a C array is fixed at compilation time.
  - **Easy and fast insertions and deletions**
    - ↳ To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
    - ↳ With a linked list, no need to move other nodes. Only need to reset some pointers.

# List Overview

## ✉ Linked lists

- It is Dynamic and so you can increase/decrease the size.
- Unlike an array which is static: can not change the size at run time

## ✉ Basic operations of linked lists

- Insert, find, delete, print, modify....

## ✉ Subset of linked lists

- **Stacks** : used in the program/thread stack: Last In First Output (L.I.F.O.)
- **Queues**: used for algorithms for scheduling processes to run on a CPU: First In First Out(F.I.F.O.)

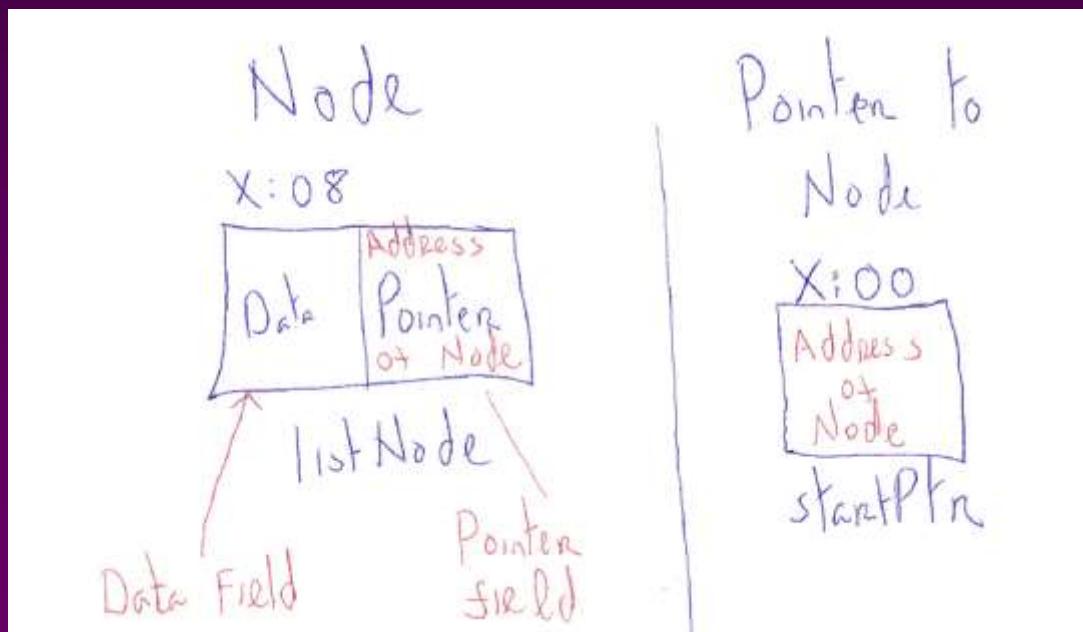
# Operations on Linked List

- ☒ // prototypes: note data types
  - ☒ void insert(ListNode\*\*, char);
  - ☒ char delete(ListNode\*, char value);
  - ☒ void printList(ListNode\*);
- 
- ☒ Pass ListNode variable by **reference** in the insert and delete function: pass the address of ListNode
  - ☒ Pass ListNode variable by **value** in the print function: pass the contents of ListNode

# The listNode structure

```
// self-referential structure
struct listNode {
    char data; // each listNode contains a character
    struct listNode *nextPtr; // pointer to next node
};

typedef struct listNode ListNode; // synonym for struct listNode
//typedef ListNode* ListNode*; // synonym for ListNode*
```



# Main function (insert node)

✉

```
int main(void)
{
    ListNode* startPtr = NULL; // initially there are no nodes
    char item; // char entered by user

    menu(); // display the menu
    printf("%s", "? ");
    unsigned int choice; // user's choice
    scanf("%u", &choice);

    // loop while user does not choose 5
    while (choice != 3) {

        switch (choice) {
            case 1:
                printf("%s", "Enter a character: ");
                scanf("\n%c", &item);

                // code to call the insert function and the print function
                insert(&startPtr, item); // insert item in list
                printList(startPtr); // print all nodes in the list

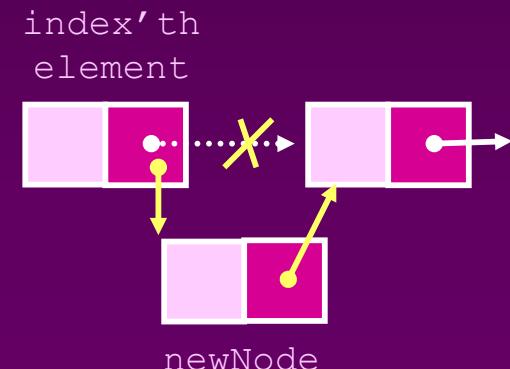
                break;
        }
    }
}
```

# Inserting a new node

- ✉ Possible cases of Insert a Node
  - 1. Insert into an empty list
  - 2. Insert in start of list
  - 3. Insert at end of list
  - 4. Insert in middle
- ✉ But, in fact, only need to handle two cases
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

# Inserting a new node

- ☒ Insert a node with data equal to  $x$  (in the Link List  
Demo the data is added in alphabetical order)
- ☒ Steps (insert function)
  1. Allocate memory for the new node
  2. Locate node (current) before which new node is inserted
  3. Point the new node to the (current) node
  4. Point the current node's **predecessor** (previous) to the new node



# the insert function

```
// insert a new value into the list in sorted order
void insert(ListNode* *sPtr, char value)
{
    ListNode* newPtr = malloc(sizeof(ListNode)); // create node

    if (newPtr != NULL) { // is space available
        newPtr->data = value; // place value in node
        newPtr->nextPtr = NULL; // node does not link to another node

        ListNode* previousPtr = NULL;
        ListNode* currentPtr = *sPtr;

        // loop to find the correct location in the list
        while (currentPtr != NULL && value > currentPtr->data) {
            previousPtr = currentPtr; // walk to ...
            currentPtr = currentPtr->nextPtr; // ... next node
        }

        // insert new node at beginning of list
        if (previousPtr == NULL) {
            newPtr->nextPtr = *sPtr;
            *sPtr = newPtr;
        }
        else { // insert new node between previousPtr and currentPtr
            previousPtr->nextPtr = newPtr;
            newPtr->nextPtr = currentPtr;
        }
    }
    else {
        printf("%c not inserted. No memory available.\n", value);
    }
}
```

# Create node and initialise pointer variable

```
[denis.manley@apollo: ~/OS2/week4]
// insert a new value into the list in sorted order
void insert(ListNode* *sPtr, char value)
{
    ListNode* newPtr = malloc(sizeof(ListNode)); // create node

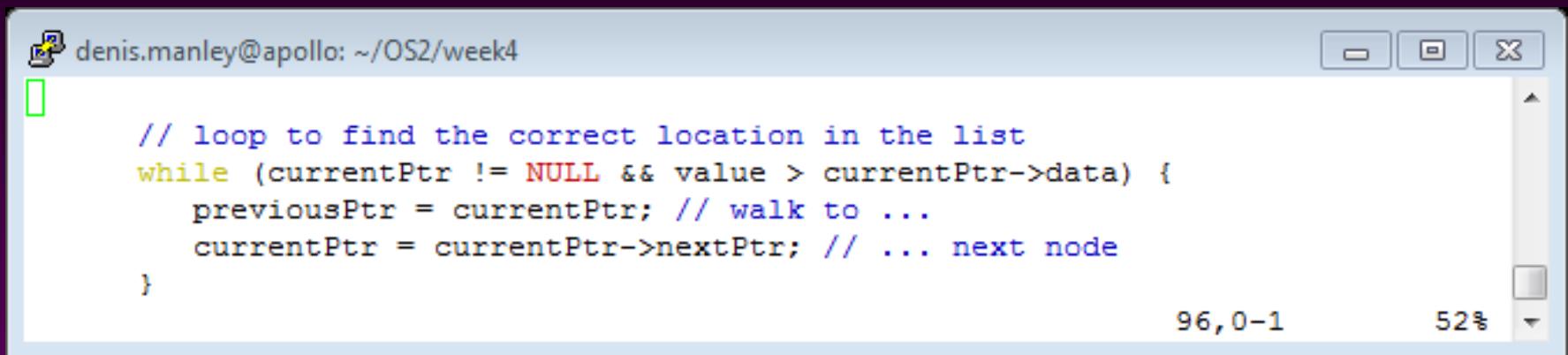
    if (newPtr != NULL) { // is space available
        newPtr->data = value; // place value in node
        newPtr->nextPtr = NULL; // node does not link to another node

        ListNode* previousPtr = NULL;
        ListNode* currentPtr = *sPtr;
```

85,1

48%

# Traversing the list

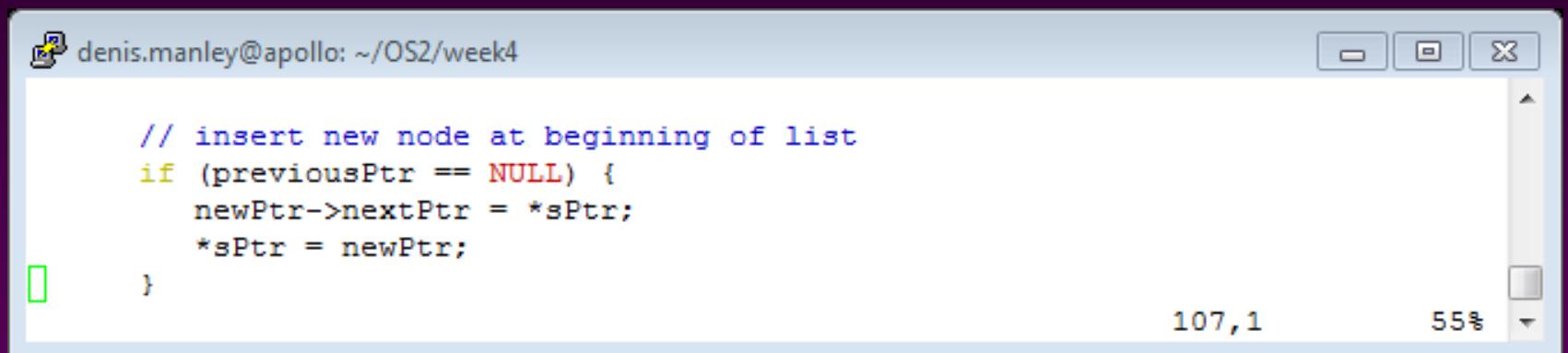


A screenshot of a terminal window titled "denis.manley@apollo: ~/OS2/week4". The window contains the following C code:

```
// loop to find the correct location in the list
while (currentPtr != NULL && value > currentPtr->data) {
    previousPtr = currentPtr; // walk to ...
    currentPtr = currentPtr->nextPtr; // ... next node
}
```

The code implements a search algorithm for a linked list. It uses three pointers: `currentPtr`, `previousPtr`, and `value`. The loop continues as long as `currentPtr` is not `NULL` and the value stored in `currentPtr` is less than or equal to `value`. Inside the loop, `previousPtr` is set to `currentPtr` and `currentPtr` is moved to the next node by setting it to `currentPtr->nextPtr`. The status bar at the bottom right shows "96, 0-1" and "52%".

# Insert a beginning of list

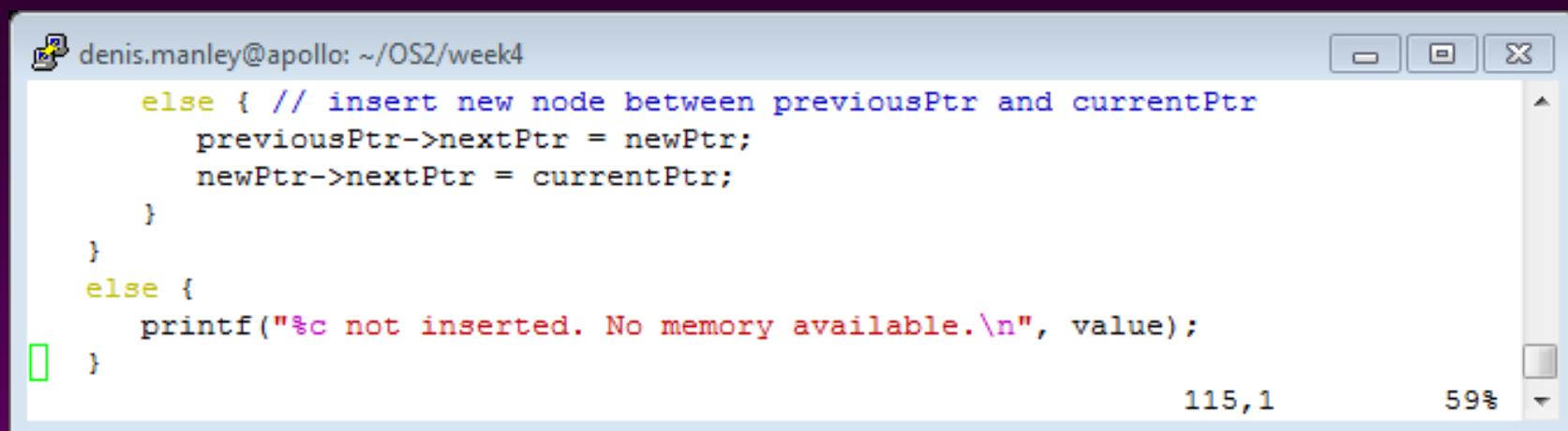


The screenshot shows a terminal window with the following details:

- Title bar: denis.manley@apollo: ~/OS2/week4
- Code content:

```
// insert new node at beginning of list
if (previousPtr == NULL) {
    newPtr->nextPtr = *sPtr;
    *sPtr = newPtr;
}
```
- Status bar: 107,1 55%

# Insert node in other part of list

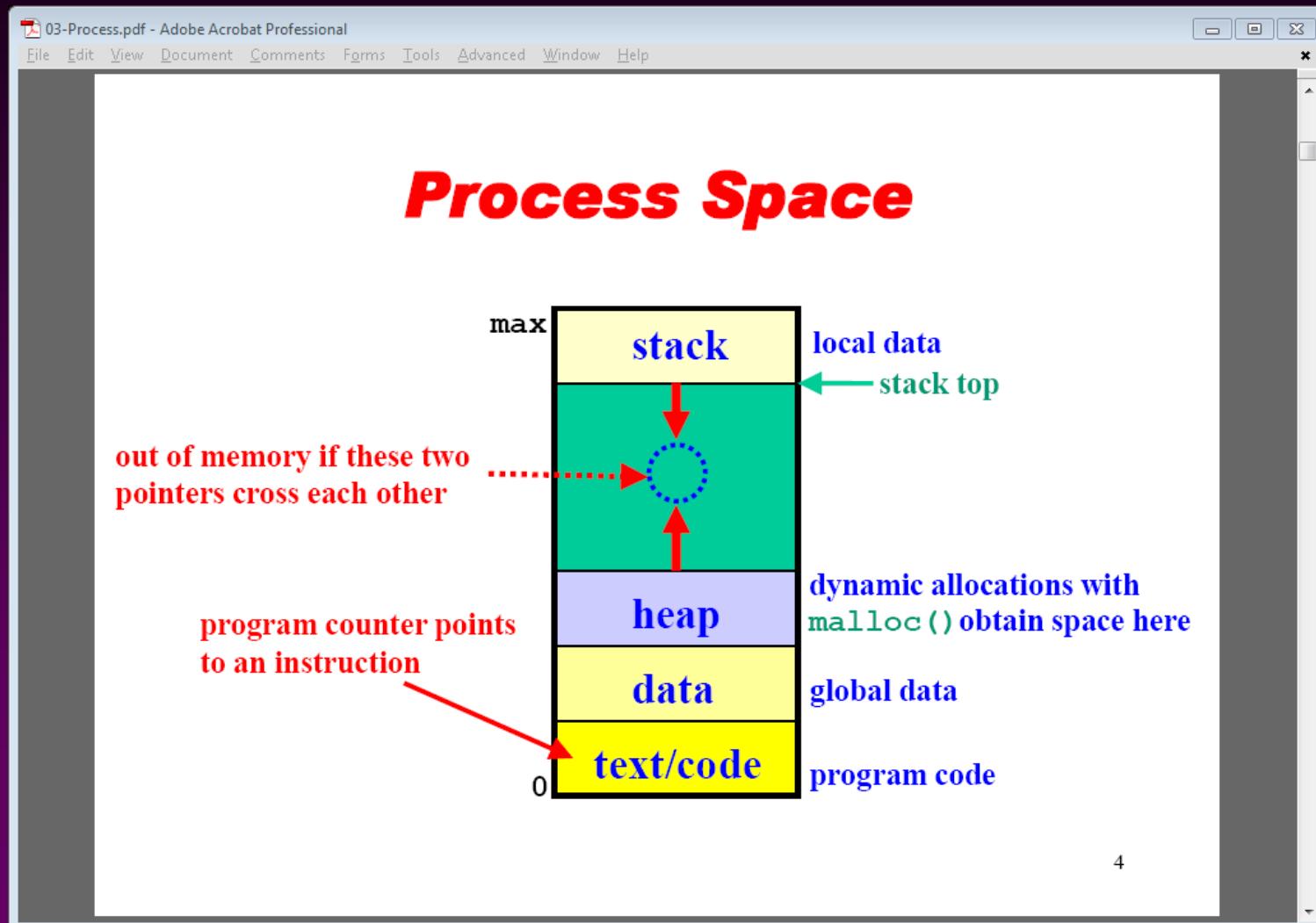


The screenshot shows a terminal window titled "denis.manley@apollo: ~/OS2/week4". The window contains the following C code:

```
else { // insert new node between previousPtr and currentPtr
    previousPtr->nextPtr = newPtr;
    newPtr->nextPtr = currentPtr;
}
else {
    printf("%c not inserted. No memory available.\n", value);
}
```

The code implements an insertion operation. It first checks if there is enough memory to create a new node. If so, it updates the pointers of the previous and current nodes to link the new node into the list. If no memory is available, it prints an error message.

# Process Space



# Example of inserting

```
denis.manley@soc-apollo:~/OS2/week4$ ./Link_List_Demo
```

```
The address of startPtr is 0x7ffe04f3c760
Enter your choice:
 1 to insert an element into the list.
 2 to delete an element from the list.
 3 to end.
```

```
? 1
```

```
Enter a character: c
```

```
the contents pf previous is (nil), the contents of current is (nil)
```

```
The list is:
```

```
c|(nil) (&0x55ef0eaa2ac0) --> NULL
```

Node “pointer” address

```
after operation the contents of startPtr is:: 0x55ef0eaa2ac0
```

```
Enter your choice:
```

- 1 to insert an element into the list.
- 2 to delete an element from the list.
- 3 to end.

```
? 1
```

```
Enter a character: a
```

```
the contents pf previous is (nil), the contents of current is 0x55ef0eaa2ac0
```

```
The list is:
```

```
a|0x55ef0eaa2ac0 (&0x55ef0eaa2ae0) --> c|(nil) (&0x55ef0eaa2ac0) --> NULL
```

Node address

Node contents

```
after operation the contents of startPtr is:: 0x55ef0eaa2ae0
```

```
Enter your choice:
```

- 1 to insert an element into the list.
- 2 to delete an element from the list.
- 3 to end.

```
? 1
```

```
Enter a character: d
```

```
value in data is a
```

```
value of currentPtr is: 0x55ef0eaa2ac0 value in data is c
```

```
value of currentPtr is: (nil) the contents pf previous is 0x55ef0eaa2ac0, the contents of current is (nil)
```

```
The list is:
```

```
a|0x55ef0eaa2ac0 (&0x55ef0eaa2ae0) --> c|0x55ef0eaa2b00 (&0x55ef0eaa2ac0) --> d|(:il) (&0x55ef0eaa2b00) --> NULL
```

Contents of variable in  
walkthrough

```
after operation the contents of startPtr is:: 0x55ef0eaa2ae0
```

# Exercise 1: insert a node empty list

- ✉ Using the variable contents and addresses from previous slide: Explain, using illustrate or otherwise:
  - What is the contents of sPtr?
  - What is contents of the variables (newPtr, currentPtr and Previous Ptr) before inserting the new node?
  - What is contents of the variables after inserting the new node D (it is an ordered link list. )

# Deleting a node

- ✉ char DeleteNode (ListNode \*\*sPtr, char x)
  - Delete a node with the value equal to x from the list.
  - If such a node is found, return its value. Otherwise, return '\0' (Null character).

## ✉ Steps

- Find the desirable node (similar to insert fnt)
- Set the pointer of the predecessor (previousPtr) of the *delete* node (currentPtr) to the successor of the found node
- Release the memory occupied by the *delete* node

## ✉ There are two cases of deletion

- Delete first node
- Delete the node in middle or at the end of the list

# Delete a node: call function

```
case 2: // delete an element
    // if list is not empty
    if (startPtr != NULL) {
        printf("%s", "Enter character to be deleted: ");
        scanf("\n%c", &item);

        // if character is found, remove it
        if (delete(&startPtr, item)) { // remove item
            printf("%c deleted.\n", item);
            printList(startPtr);
        }
        else {
            printf("%c not found.\n\n", item);
        }
    }
    else {
        puts("List is empty.\n");
    }

    break;
```

# Delete function

```
☒ // delete a list element
char delete(ListNode* *sPtr, char value)
{
    // delete first node if a match is found
    if (value == (*sPtr)->data) {
        ListNode* tempPtr = *sPtr; // hold onto node being removed
        *sPtr = (*sPtr)->nextPtr; // de-thread the node
        free(tempPtr); // free the de-threaded node
        return value;
    }
    else {
        ListNode* previousPtr = *sPtr;
        ListNode* currentPtr = (*sPtr)->nextPtr;

        // loop to find the correct location in the list
        while (currentPtr != NULL && currentPtr->data != value) {
            previousPtr = currentPtr; // walk to ...
            currentPtr = currentPtr->nextPtr; // ... next node
        }

        // delete node at currentPtr
        if (currentPtr != NULL) {
            ListNode* tempPtr = currentPtr;
            previousPtr->nextPtr = currentPtr->nextPtr;
            free(tempPtr);
            return value;
        }
    }
    return '\0'; // '\0' is the null character (equivalent to returning false)
}
```

# Sample output add 4 nodes

```
denis.manley@apollo:~/OS2/week4$ ./Linklist_demo
Enter your choice:
  1 to insert an element into the list.
  2 to delete an element from the list.
  3 to end.
? 1
before adding/deleting the address of startPtr is 0x7ffe6c886a68; the contents are: (nil)
Enter a character: C
the address of the new node is 0xc7e010; the contents of the data field is: C
The list is:
C | (nil) --> NULL

after operation: the address of startPtr is 0x7ffe6c886a68; the contents are: 0xc7e010
Enter your choice:
  1 to insert an element into the list.
  2 to delete an element from the list.
  3 to end.
? 1
Enter a character: A
the address of the new node is 0xc7e030; the contents of the data field is: A
The list is:
A | 0xc7e010 --> C | (nil) --> NULL

after operation: the address of startPtr is 0x7ffe6c886a68; the contents are: 0xc7e030
Enter your choice:
  1 to insert an element into the list.
  2 to delete an element from the list.
  3 to end.
? 1
Enter a character: B
the address of the new node is 0xc7e050; the contents of the data field is: B
The list is:
A | 0xc7e050 --> B | 0xc7e010 --> C | (nil) --> NULL

after operation: the address of startPtr is 0x7ffe6c886a68; the contents are: 0xc7e030
Enter your choice:
  1 to insert an element into the list.
  2 to delete an element from the list.
  3 to end.
? 1
Enter a character: D
the address of the new node is 0xc7e070; the contents of the data field is: D
The list is:
A | 0xc7e050 --> B | 0xc7e010 --> C | 0xc7e070 --> D | (nil) --> NULL
```

# Sample output delete 3 nodes

```
after operation the contents of startPtr is:: 0x55b4cf17aae0
Enter your choice:
  1 to insert an element into the list.
  2 to delete an element from the list.
  3 to end.
? 2
Enter character to be deleted: a
a deleted.
The list is:
b|0x55b4cf17aac0 (&0x55b4cf17ab00) --> c|0x55b4cf17ab20 (&0x55b4cf17aac0) --> d|(nil) (&0x55b4cf17ab20) --> NULL

after operation the contents of startPtr is:: 0x55b4cf17ab00
Enter your choice:
  1 to insert an element into the list.
  2 to delete an element from the list.
  3 to end.
? 2
Enter character to be deleted: c
the contents pf previous is 0x55b4cf17ab00, the contents of current is 0x55b4cf17aac0
c deleted.
The list is:
b|0x55b4cf17ab20 (&0x55b4cf17ab00) --> d|(nil) (&0x55b4cf17ab20) --> NULL

after operation the contents of startPtr is:: 0x55b4cf17ab00
Enter your choice:
  1 to insert an element into the list.
  2 to delete an element from the list.
  3 to end.
? 2
Enter character to be deleted: d
the contents pf previous is 0x55b4cf17ab00, the contents of current is 0x55b4cf17ab20
d deleted.
The list is:
b|(nil) (&0x55b4cf17ab00) --> NULL
```

# Exercise 2: delete nodes from a list

- ✉ Using the variable contents and addresses from previous slide to Explain using illustrations or otherwise:
  - What is the value (not the address) of sPtr?
  - What is contents of the variables (newPtr, currentPtr and Previous Ptr) before inserting the new node?
  - What is contents of the variable after deleting a node from the list (Assume the ordered list contains 4 nodes... )

# Print Function

- ✉ Call the print function (use a pointer to a node node address of pointer to a node):

```
printList(startPtr);
```

- ✉ Print function definition

```
// print the list
void printList(ListNode* currentPtr)
{
    // if list is empty
    if (currentPtr == NULL) {
        puts("List is empty.\n");
    }
    else {
        puts("The list is:");

        // while not the end of the list
        while (currentPtr != NULL) {
            printf("%c --> ", currentPtr->data);
            currentPtr = currentPtr->nextPtr;
        }

        puts("NULL\n");
    }
}
```

# Exercise 3: print nodes in a list

✉ Using the variable contents and addresses Explain using illustrations or otherwise to illustrate:

- What is the initial value (not the address) of currentPtr?
- What is contents of the field (currentPtr->nextPtr)?
- What is contents of these variables after printing the first node (One iteration of the While loop)

the contents of startPtr is::: 0x5a3fd344bae0

The list is:  
a|0x5a3fd344bac0 (&0x5a3fd344bae0) --> c|(nil) (&0x5a3fd344bac0) --> NULL

# homework

- ✉ Write a function that will change the data contents of a node:
  - Prompt the user for the ID
  - Pass to a function to find a node
  - If found change contents. Print the list with the edited node
  - If not found return an appropriate message

# Sample Exam Question

- ✉ Why is a link list a more efficient way of utilisation of memory than arrays **(3 marks)**
  
- ✉ Explain, using a simple example, how the add/delete functions , in a link list program, use pointers to keep tract of any changes made to the pointer to the beginning of the list (Head) **(5 marks)**

# Sample Question

- ✉ Clearly explain, using suitable examples, the steps required to:
  - Add a node into an ordered link list: at the start at the middle and at the end **(10 marks)**
  - Delete a node from an ordered link list, the start, middle and end of a link list **(10 marks)**

Clearly explain if the following code will **delete** a node from a link list (10 marks)

```
// delete a list element
char delete(ListNode* *sPtr, char value)
{
    // delete first node if a match is found
    if (value == (*sPtr)->data) {
        ListNode* tempPtr = *sPtr; // hold onto node being removed
        sPtr = (*sPtr)->nextPtr; // de-thread the node
        free(tempPtr); // free the de-threaded node
        return value;
    }
    else {
        ListNode* previousPtr = *sPtr;
        ListNode* currentPtr = (*sPtr)->nextPtr;

        // loop to find the correct location in the list
        while (currentPtr != NULL && currentPtr->data != value) {
            previousPtr = currentPtr; // walk to ...
            currentPtr = currentPtr->nextPtr; // ... next node
        }

        // delete node at currentPtr
        if (currentPtr != NULL) {
            ListNode* tempPtr = currentPtr;
            previousPtr->nextPtr = currentPtr->nextPtr;
            free(tempPtr);
            return value;
        }
    }

    return '\0'; // '\0' is the null character (equivalent to returning false)
}
```

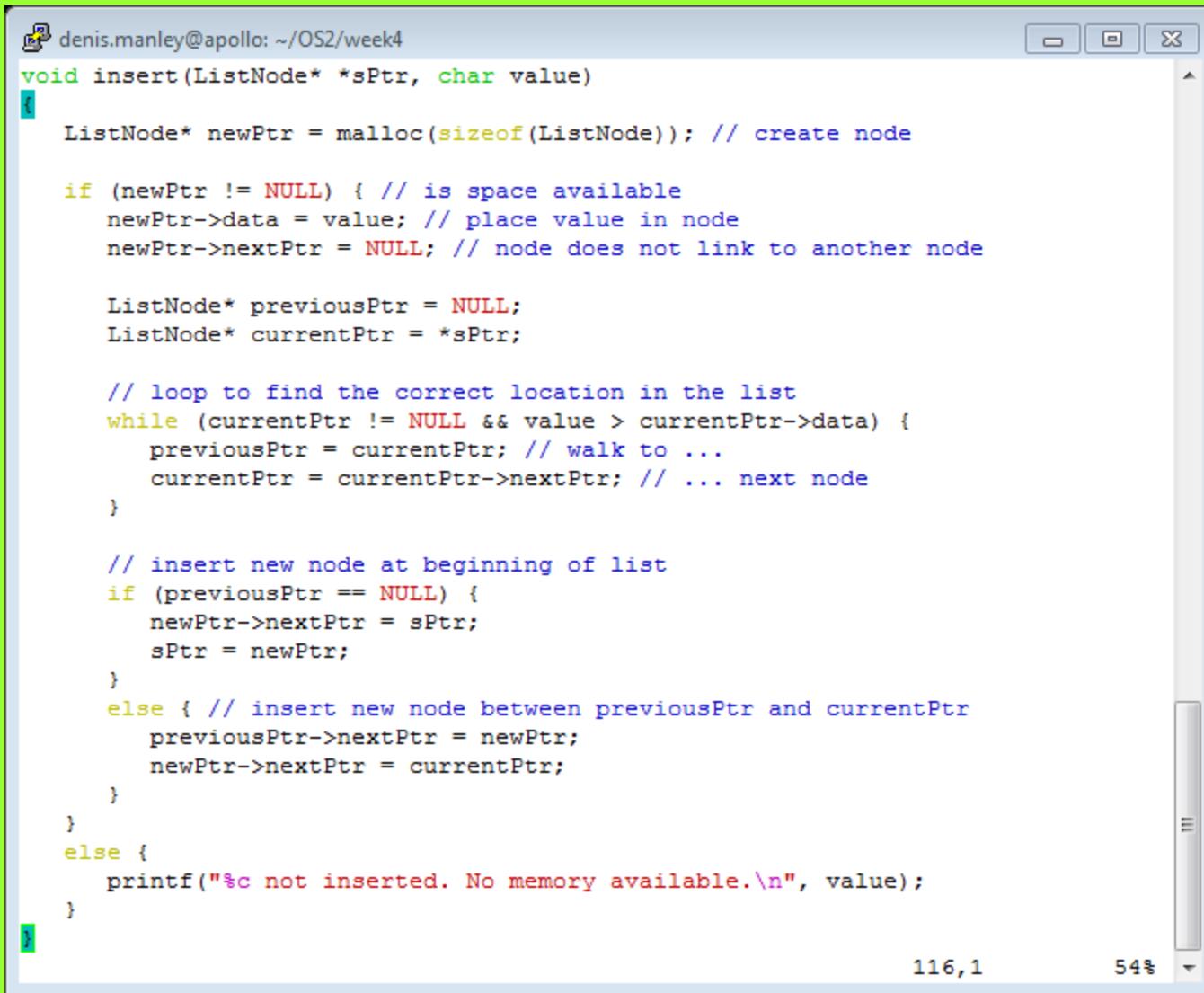
Clearly explain if the following code will **delete** a node from a link list (10 marks)

```
// delete a list element
char delete(ListNode* *sPtr, char value)
{
    // delete first node if a match is found
    if (value == (*sPtr)->data) {
        ListNode* tempPtr = *sPtr; // hold onto node being removed
        *sPtr = (*sPtr)->nextPtr; // de-thread the node
        free(tempPtr); // free the de-threaded node
        return value;
    }
    else {
        ListNode* previousPtr = *sPtr;
        ListNode* currentPtr = (*sPtr)->nextPtr;

        // loop to find the correct location in the list
        while (currentPtr != NULL && currentPtr->data != value) {
            previousPtr = currentPtr; // walk to ...
            currentPtr = currentPtr->nextPtr; // ... next node
        }

        // delete node at currentPtr
        if (currentPtr != NULL) {
            ListNode* tempPtr = currentPtr;
            previousPtr->nextPtr = currentPtr->nextPtr;
            free(tempPtr);
            return value;
        }
    }
    return '\0'; // '\0' is the null character (equivalent to returning false)
}
```

Clearly explain if the following code will **insert** a node onto a link list (10 marks)  
note the comments will be removed in the exam



```
denis.manley@apollo: ~/OS2/week4
void insert(ListNode* *sPtr, char value)
{
    ListNode* newPtr = malloc(sizeof(ListNode)); // create node

    if (newPtr != NULL) { // is space available
        newPtr->data = value; // place value in node
        newPtr->nextPtr = NULL; // node does not link to another node

        ListNode* previousPtr = NULL;
        ListNode* currentPtr = *sPtr;

        // loop to find the correct location in the list
        while (currentPtr != NULL && value > currentPtr->data) {
            previousPtr = currentPtr; // walk to ...
            currentPtr = currentPtr->nextPtr; // ... next node
        }

        // insert new node at beginning of list
        if (previousPtr == NULL) {
            newPtr->nextPtr = sPtr;
            sPtr = newPtr;
        }
        else { // insert new node between previousPtr and currentPtr
            previousPtr->nextPtr = newPtr;
            newPtr->nextPtr = currentPtr;
        }
    }
    else {
        printf("%c not inserted. No memory available.\n", value);
    }
}
```

116,1

54%

# Clearly explain if the following code will **insert** a node onto a link list (10 marks) note the comments will be removed in the exam

```
// insert a new value into the list in sorted order
void insert(ListNode* *sPtr, char value)
{
    char continu;
    ListNode* newPtr = malloc(sizeof(ListNode)); // create node

    if (newPtr != NULL) { // is space available
        newPtr->data = value; // place value in node
        newPtr->nextPtr = NULL; // node does not link to another node
        printf("the address of the new node is %p; the contents of the data field is: %c\n", newPtr, newPtr->data);

        ListNode* previousPtr = NULL;
        ListNode* currentPtr = *sPtr;

        // loop to find the correct location in the list
        while (currentPtr != NULL && value > currentPtr->data) {
            previousPtr = currentPtr; // walk to ...
            currentPtr = currentPtr->nextPtr; // ... next node
            //printf("previous pointer %p \t current pointer %p press return...\n", previousPtr, currentPtr);
            //scanf("%c", &continu);
        }

        // insert new node at beginning of list
        if (previousPtr == NULL) {
            newPtr->nextPtr = *sPtr;
            *sPtr = newPtr;
        }
        else { // insert new node between previousPtr and currentPtr
            previousPtr->nextPtr = newPtr;
            newPtr->nextPtr = currentPtr;
        }
    }
    else {
        printf("%c not inserted. No memory available.\n", value);
    }
}
```