MANCHESTER 1824
The University of Manchester

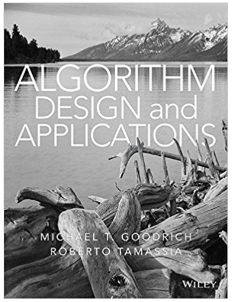
COMP26120: Linked List in C (2019/20)

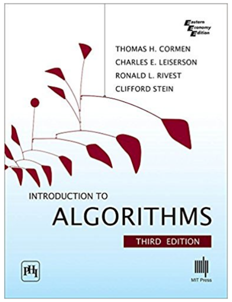
Lucas Cordeiro
lucas.cordeiro@manchester.ac.uk

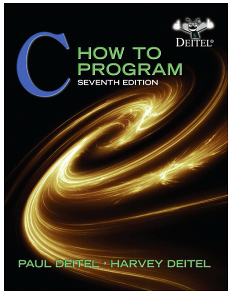
Linked List

- Lucas Cordeiro (Formal Methods Group)
  - *lucas.cordeiro@manchester.ac.uk*
  - Office: 2.28
  - Office hours: 15-16 Tuesday, 14-15 Wednesday
- Textbook:
  - *Algorithm Design and Applications* (Chapter 2)
  - *Introduction to Algorithms* (Chapter 10)
  - *C How to Program* (Chapter 12)

These slides are based on the lectures notes of "C How to Program"







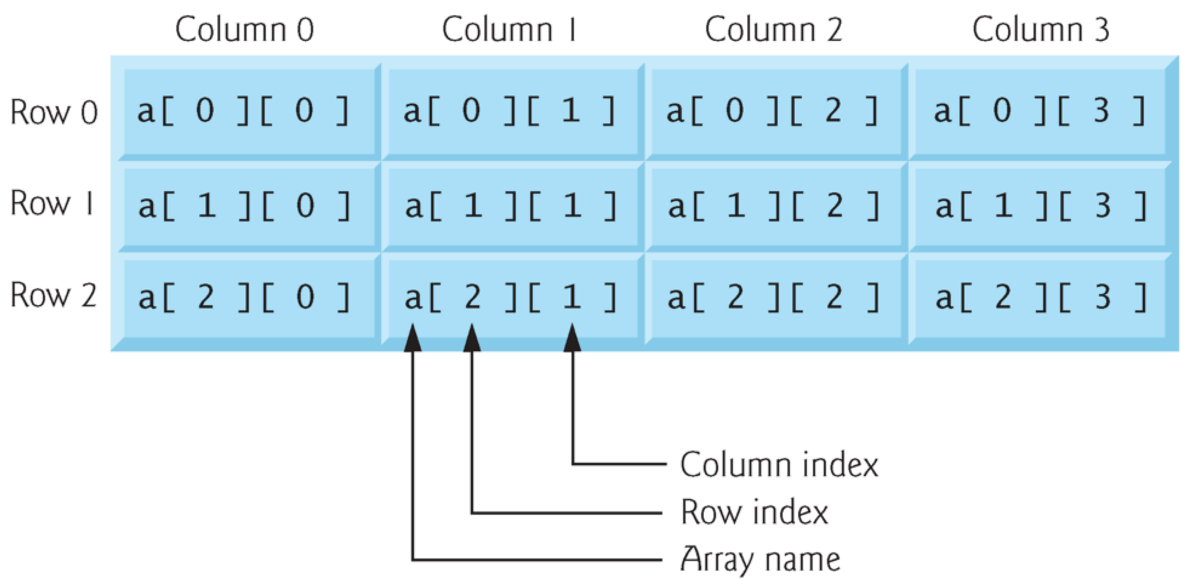
Intended learning outcomes

- To be able to **allocate** and **free** memory dynamically for data objects
- To form linked data structures using **pointers**, **self-referential structures** and **recursion**
- To be able to create and manipulate **linked lists**
- To understand various important **applications** of **linked data structures**
- To study **secure C programming practices** for pointers and dynamic memory allocation

Introduction

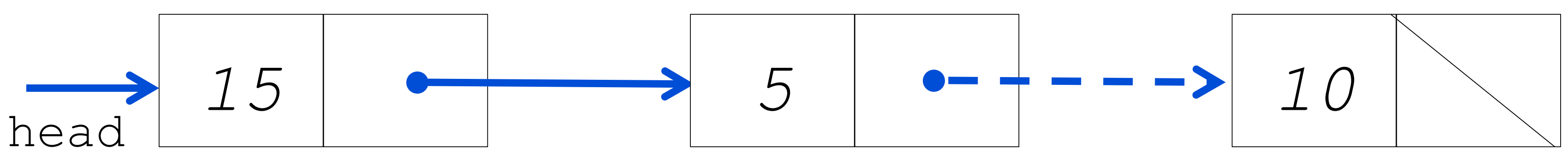
- We've studied fixed-size data structures such as **single-subscripted arrays**, **double-subscripted arrays** and `structs`

```
typedef struct account {
  unsigned short age;
  char name[100];
} accountt;

int main()
{
  int x[3];
  int a[3][4];
  accountt acount;
  return 0;
}
```



Introduction

- We've studied fixed-size data structures such as **single-subscripted arrays**, **double-subscripted arrays** and `structs`
- Dynamic data structures
  - They can **grow** and **shrink** during execution
- Linked lists
  - Allow **insertions** and **removals** anywhere in a linked list



Self-referential structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Terminated with a NULL pointer (**0**)

```
typedef struct node {
   int data;
   struct node *nextPtr;
} nodet;
```

Not setting the link in the last node of a list to NULL can lead to runtime errors

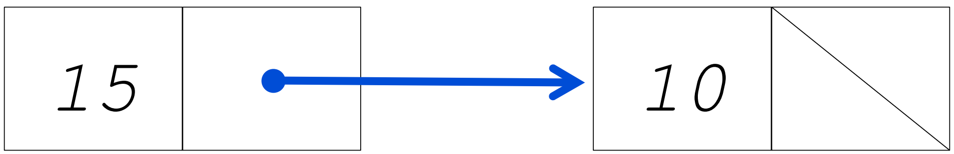
  - `nextPtr`
    - Points to an object of type node
    - Referred to as a link
    - Ties one **node** to another **node**
  - Can be linked together to form useful data structures such as **lists**, **queues**, **stacks** and **trees**

Dynamic memory allocation

- Dynamic memory allocation
  - **Obtain** and **release** memory during execution
- `malloc`
  - Takes number of bytes to allocate
    - Use `sizeof` to determine the size of an object
  - Returns pointer of type `void *`
    - A `void *` pointer may be assigned to any pointer
    - If no memory available, returns `NULL`
  - Example: `nodet *newPtr = (nodet *)malloc(sizeof(nodet));`
- `free`
  - Always deallocates memory allocated by `malloc` to avoid **memory leak**
  - Takes a pointer as an argument
    - `free (newPtr);`

Dynamic memory allocation

Two self-referential structures linked together

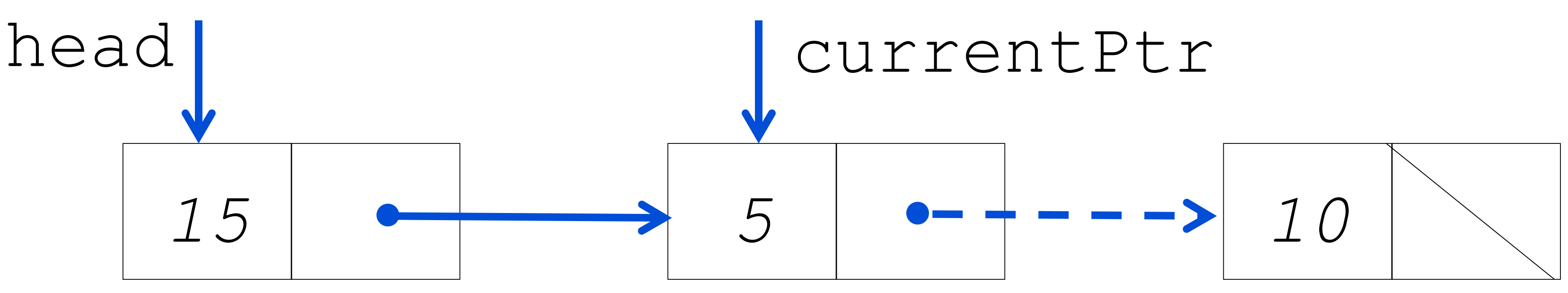



```
int main() {
  // allocates memory
  nodet *node1 = (nodet *)malloc(sizeof(nodet));
  nodet *node2 = (nodet *)malloc(sizeof(nodet));
  node1->data = 15;
  node2->data = 10;
  // link node1 to node2
  node1->nextPtr = node2;
  node2->nextPtr = NULL;
  // Deallocates memory allocated by malloc
  free(node1);
  free(node2);
  return 0;
}
```

If there exists no memory available, then `malloc` returns `NULL`

Linked lists properties

- Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the link-pointer member of the current node
  - Link pointer in the last node is set to NULL to mark the list's end

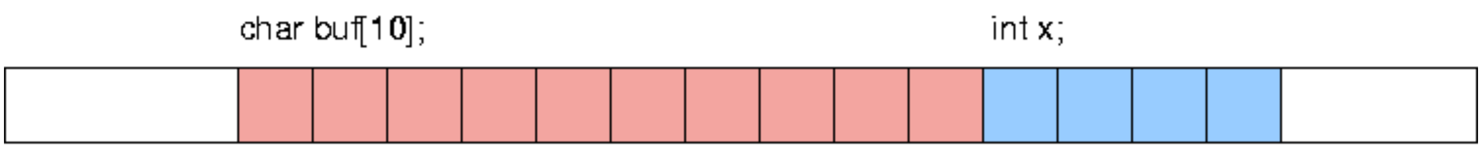


Linked lists properties

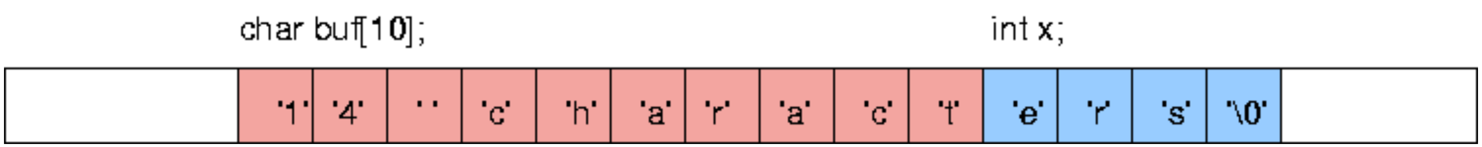
- Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the link-pointer member of the current node
  - Link pointer in the last node is set to NULL to mark the list's end
- Use a **linked list** instead of an **array** when
  - You have an **unpredictable** number of elements
  - Your list needs to be **sorted quickly**

Linked lists properties

- Linked lists are **dynamic**, so the length of a list can **increase** or **decrease** as necessary
- Can we change the array size after compiling the program? What are the problems here?
  - **Arrays** can become **full**
    - An array can be declared to contain more elements than the number of data items expected, but this can waste memory



strcpy (buf, "14 characters");



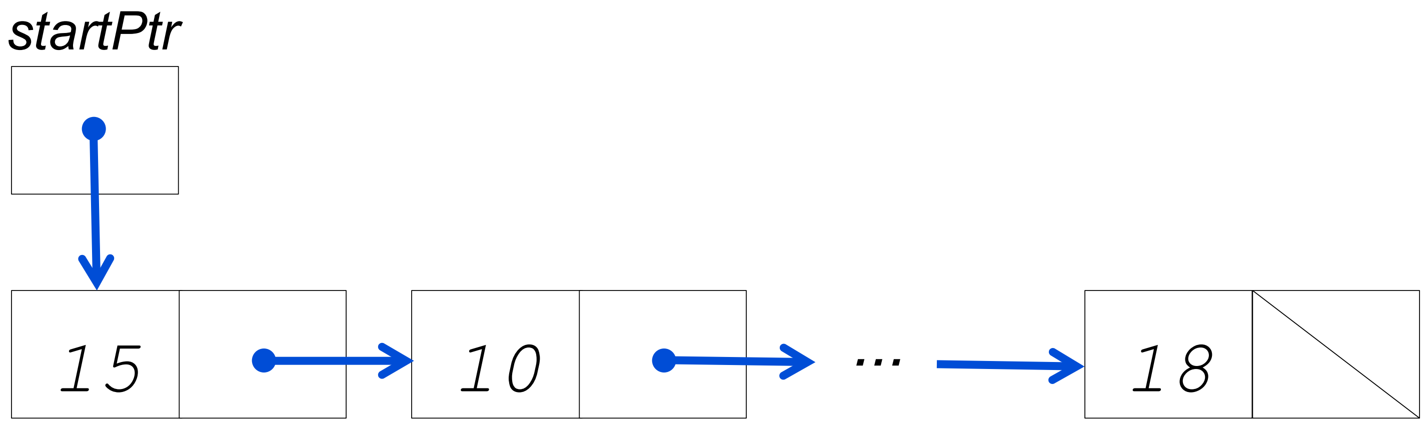
Linked lists properties

- Linked lists are **dynamic**, so the length of a list can **increase** or **decrease** as necessary
- Can we change the array size after compiling the program? What are the problems here?
  - **Arrays** can become **full**
    - An array can be declared to contain more elements than the number of data items expected, but this can waste memory
- **Linked lists** become **full** only when the system has **insufficient memory** to satisfy dynamic storage allocation requests
  - It can provide better **memory utilization**

Linked lists properties

- Linked-list nodes are **normally not stored contiguously in memory**
  - How arrays are stored in memory? What would be the advantage here?
    - This allows **immediate access** since the address of any element can be calculated directly based on its position relative to the beginning of the array
      - Linked lists do not afford such immediate access
- Logically, however, the nodes of a linked list appear to be contiguous
  - Pointers take up **space** and dynamic memory allocation incurs the **overhead of function calls**

A graphical representation of a linked list




```
int main() {
   …
   // link the nodes
   startPtr = node1;
   node1->nextPtr = node2;
   node2->nextPtr = node3;
   node3->nextPtr = NULL;
   …
   return 0;
}
```

Pointers should be initialised before they're used

A structure's size is not necessarily the sum of the size of its members (machine-dependent boundary alignment)

Error prevention when using linked lists

- If dynamically allocated memory is no longer needed, use **`free`** to return it to the system
  - Why must we set that pointer to `NULL`?
    - eliminate the possibility that the program could refer to memory that's been reclaimed and which may have already been allocated for another purpose
- Is it an error to free memory not allocated dynamically with `malloc`?
  - Referring to memory that has been freed is an error, which results in the program crashing **(double free)**

Exercise

- Fill in the blanks in each of the following:

a) A self- _______ structure is used to form dynamic data structures.

b) Function ______ is used to dynamically allocate memory.

c) A(n) ______ is a specialized version of a linked list in which nodes can be inserted and deleted only from the start of the list.

d) Functions that look at a linked list but do not modify it are referred to as _________.

e) Function ______ is used to reclaim dynamically allocated memory.

Illustrative example about linked lists

- We will show an example of linked list that manipulates a list of characters
- You can insert a character in the list in alphabetical order (function `insert`) or to delete a character from the list (function `delete`)

```c
// Fig. 12.3: fig12_03.c
// Inserting and deleting nodes in a list
#include <stdio.h>
#include <stdlib.h>

// 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 *ListNodePtr; // synonym for ListNode*

// prototypes
void insert(ListNodePtr *sPtr, char value);
char delete(ListNodePtr *sPtr, char value);
int isEmpty(ListNodePtr sPtr);
void printList(ListNodePtr currentPtr);
void instructions(void);

int main(void)
{
```

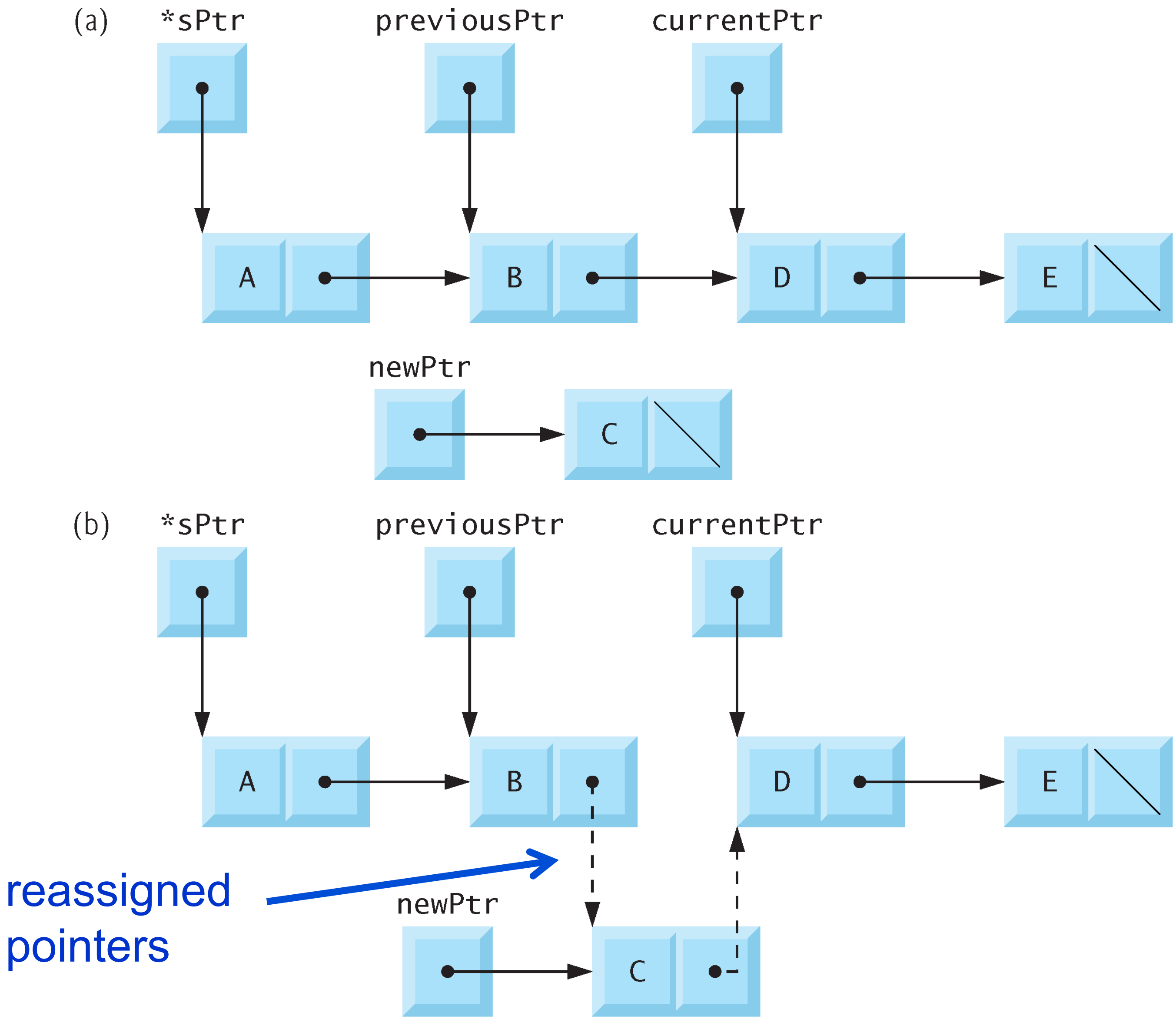
Inserting and deleting nodes in a list (Part 1 of 8)

```
    ListNodePtr startPtr = NULL; // initially there are no nodes
    char item; // char entered by user

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

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

        switch (choice) {
            case 1:
                printf("%s", "Enter a character: ");
                scanf("\n%c", &item);
                insert(&startPtr, item); // insert item in list
                printList(startPtr);
                break;
            case 2: // delete an element
                // if list is not empty
                if (!isEmpty(startPtr)) {
                    printf("%s", "Enter character to be deleted: ");
                    scanf("\n%c", &item);

```

Inserting and deleting nodes in a list (Part 2 of 8)

```c
               // 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;
         default:
            puts("Invalid choice.\n");
            instructions();
            break;
      }

      printf("%s", "? ");
      scanf("%u", &choice);
   }

```

Inserting and deleting nodes in a list (Part 3 of 8)



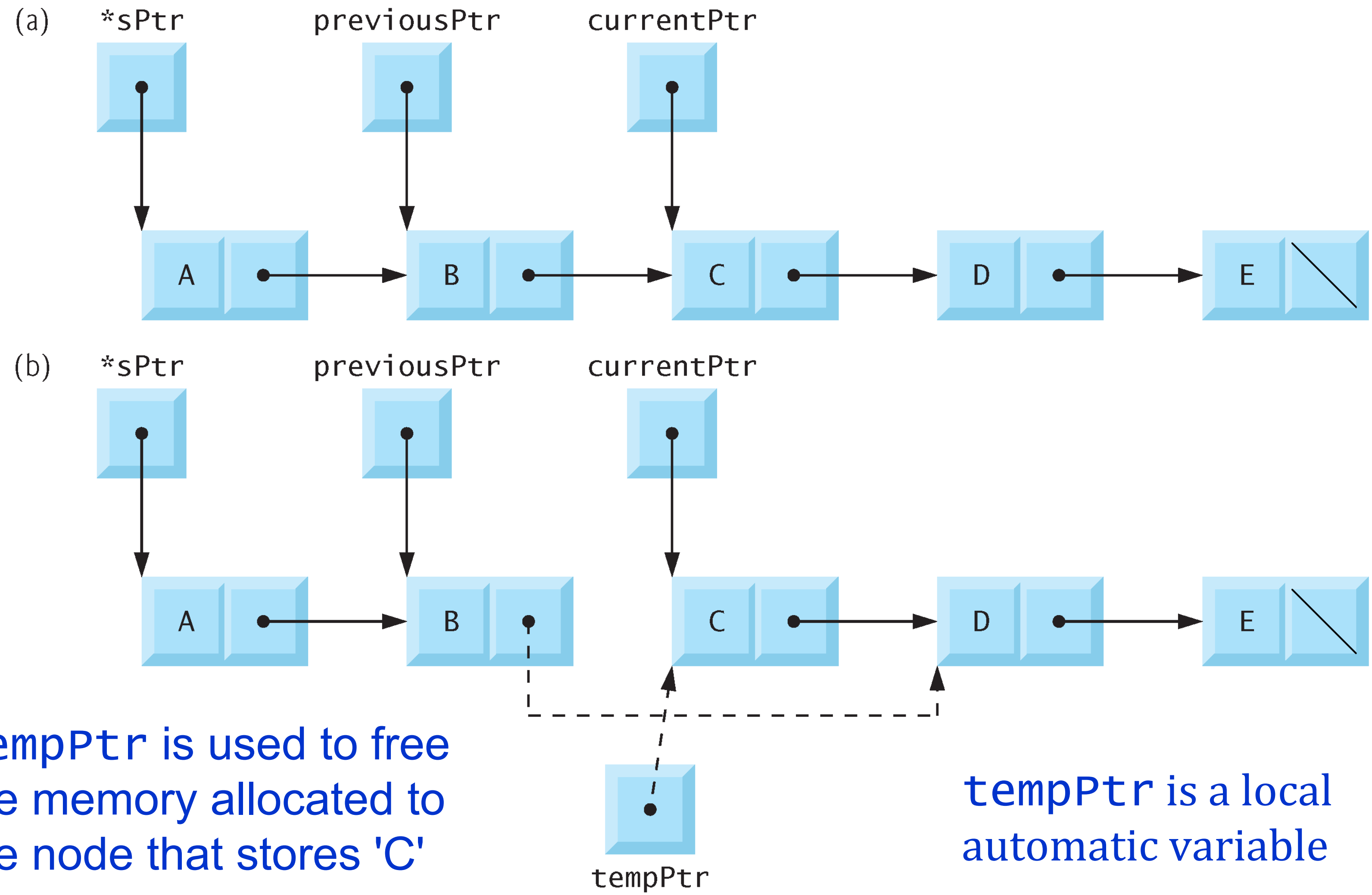
Inserting a node in order in a list

```
   puts("End of run.");
}

// display program instructions to user
void instructions(void)
{
   puts("Enter your choice:\n"
      "   1 to insert an element into the list.\n"
      "   2 to delete an element from the list.\n"
      "   3 to end.");
}

// insert a new value into the list in sorted order
void insert(ListNodePtr *sPtr, char value)
{
   ListNodePtr 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

      ListNodePtr previousPtr = NULL;
      ListNodePtr currentPtr = *sPtr;

```

Inserting and deleting nodes in a list (Part 4 of 8)

```
      // 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);
   }
}
```

Inserting and deleting nodes in a list (Part 5 of 8)



tempPtr is used to free the memory allocated to the node that stores 'C'

tempPtr is a local automatic variable

Deleting a node from a list

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

```

Inserting and deleting nodes in a list (Part 6 of 8)

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

      return '\0';
}

// return 1 if the list is empty, 0 otherwise
int isEmpty(ListNodePtr sPtr)
{
   return sPtr == NULL;
}

```

Inserting and deleting nodes in a list (Part 7 of 8)

```
// print the list
void printList(ListNodePtr currentPtr)
{
   // if list is empty
   if (isEmpty(currentPtr)) {
      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");
   }
}
```

Inserting and deleting nodes in a list (Part 8 of 8)

```
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 list is:
B --> NULL

? 1
Enter a character: A

The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.
```

Sample output for the program (Part 1 of 2)

```
? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL

? 2
Enter character to be deleted: C
C deleted.
The list is:
A --> NULL

? 2
Enter character to be deleted: A
A deleted.
List is empty.

? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.
```

Sample output for the program (Part 2 of 2)

Analysis of the linked list

OPERATION	RUNTIME (Big-O)
add to start of list	
add to end of list	
add at given index	
find an object	
remove first element	
remove last element	
remove at given index	
size	

```c
    puts("End of run.");
}

// display program instructions to user
void instructions(void)
{
    puts("Enter your choice:\n"
        "    1 to insert an element into the list.\n"
        "    2 to delete an element from the list.\n"
        "    3 to end.");
}

// insert a new value into the list in sorted order
void insert(ListNodePtr *sPtr, char value)
{
    ListNodePtr 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

        ListNodePtr previousPtr = NULL;
        ListNodePtr currentPtr = *sPtr;

```

O(1)

Analysis of the linked list (insert) – Part 1 of 2

```c
      // 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);
   }
}
```

O(n)

O(1)

Analysis of the linked list (insert) – Part 2 of 2

Insert -- runtime: O(1)+O(n)+O(1) = O(n)

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

```

O(1)

O(1)

O(n)

Analysis of the linked list (delete) – Part 1 of 2

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

   return '\0';
}

// return 1 if the list is empty, 0 otherwise
int isEmpty(ListNodePtr sPtr)
{
   return sPtr == NULL;
}

```

O(1)

Analysis of the linked list (delete) – Part 2 of 2

Delete -- runtime: O(1)+O(n)+O(1) = O(n)

Analysis of the linked list

OPERATION	RUNTIME (Big-O)
add to start of list	O(*1*)
add to end of list	O(n)
add at given index	O(*n*)
find an object	O(*n*)
remove first element	O(*1*)
remove last element	O(n)
remove at given index	O(*n*)
size	O(1)

Secure C Programming

Chapter 8 of the CERT Secure C Coding Standard

- Chapter 8 of the CERT Secure C Coding Standard is dedicated to memory-management recommendations and rules—many apply to the uses of pointers and dynamic-memory allocation presented in this chapter.
- For more information, visit www.securecoding.cert.org.

Summary
(Secure C Programming)

- Pointers should not be left uninitialized
- They should be assigned either NULL or the address of a valid item in memory
- When you use free to deallocate dynamically allocated memory, the pointer passed to free is not assigned a new value, so it still points to the memory location where the dynamically allocated memory used to be

Summary
(Secure C Programming)

- Using a pointer that's been freed can lead to program crashes and security vulnerabilities
- When you free dynamically allocated memory, you should immediately assign the pointer either NULL or a valid address
- We chose not to do this for local pointer variables that immediately go out of scope after a call to free

Summary
(Secure C Programming)

- Undefined behavior occurs when you attempt to use free to deallocate dynamic memory that was already deallocated—this is known as a “double free vulnerability”
- To ensure that you don’t attempt to deallocate the same memory more than once, immediately set a pointer to NULL after the call to free—attempting to free a NULL pointer has no effect

Summary
(Secure C Programming)

- Function malloc returns NULL if it’s unable to allocate the requested memory
- You should always ensure that malloc did not return NULL before attempting to use the pointer that stores malloc’s return value