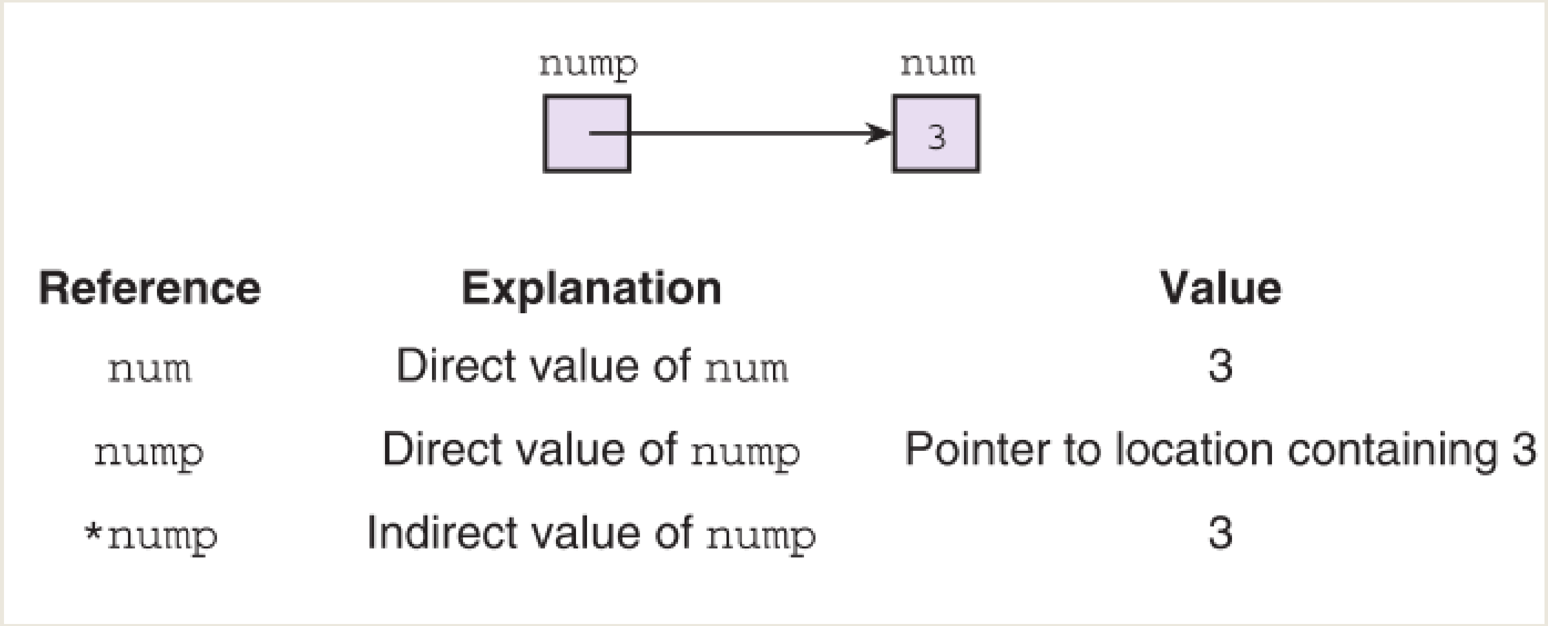
DYNAMIC DATA STRUCTURES

CEN116 Algorithms and Programming II

Terminology

- dynamic data structure
  - *a structure tat can expand and contract as a program executes*
- nodes
  - *dynamically allocated structures that are linked together to form a composite sructure*

Background



Reference	Explanation	Value
num	Direct value of num	3
nump	Direct value of nump	Pointer to location containing 3
*nump	Indirect value of nump	3

Dynamic Memory Allocation

- Creating and maintaining dynamic data structures requires **dynamic memory allocation**—the ability for a program to obtain more memory space at execution time to hold new nodes, and to release space no longer needed.
- Functions **malloc** and **free**, and operator **sizeof**, are essential to dynamic memory allocation.

Dynamic Memory Allocation

- Function **malloc** takes as an argument the number of bytes to be allocated and returns a pointer of type **void * (pointer to void)** to the allocated memory.
- As you recall, a **void *** pointer may be assigned to a variable of any pointer type.
- Function **malloc** is normally used with the **sizeof** operator.

Dynamic Memory Allocation

- For example, the statement

```
newPtr = malloc(sizeof(int));
```

- evaluates sizeof(int) to determine the size in bytes of an integer, allocates a new area in memory of that number of bytes and stores a pointer to the allocated memory in variable newPtr.
- The allocated memory is not initialized.
- If no memory is available, **malloc** returns **NULL**.

Dynamic Memory Allocation

- Function **free** deallocates memory—i.e., the memory is returned to the system so that it can be reallocated in the future.
- To **free** memory dynamically allocated by the preceding malloc call, use the statement

```
free(newPtr);
```

- C also provides functions **calloc** and **realloc** for creating and modifying **dynamic arrays**.

Dynamic Data Structures

- We’ve studied fixed-size data structures such as single-subscripted arrays, double-subscripted arrays and structs.
- This chapter introduces dynamic data structures with sizes that grow and shrink at execution time.
  - ***Linked lists*** *are collections of data items “lined up in a row”—insertions and deletions are made anywhere in a linked list.*
  - ***Stacks*** *are important in compilers and operating systems—insertions and deletions are made only at one end of a stack—its top.*

Dynamic Data Structures

- – ***Queues*** *represent waiting lines; insertions are made only at the back (also referred to as the tail) of a queue and deletions are made only from the front (also referred to as the head) of a queue.*
- – ***Binary trees*** *facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representing file system directories and compiling expressions into machine language.*

- Each of these data structures has many other interesting applications.

Self-Referential Structures

- Recall that a self-referential structure contains a pointer member that points to a structure of the same structure type.
- For example, the definition

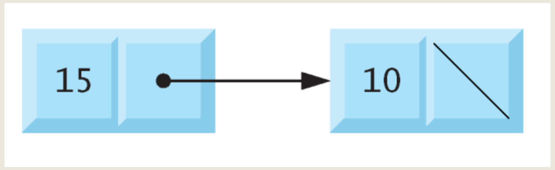
```
struct node {
   int data;
   struct node *nextPtr;
};
```

  defines a type, struct node.
- A structure of type struct node has two members—integer member data and pointer member nextPtr.

Self-Referential Structures

- Member nextPtr points to a structure of type struct node—a structure of the same type as the one being declared here, hence the term **“self-referential structure.”**
- Member nextPtr is referred to as a link—i.e., it can be used to “tie” a structure of type struct node to another structure of the same type.
- Self-referential structures can be linked together to form useful data structures such as lists, queues, stacks and trees.

Self-Referential Structures



- Figure illustrates two self-referential structure objects linked together to form a list.
- A slash—representing a NULL pointer—is placed in the link member of the second self-referential structure to indicate that the link does not point to another structure.
- A NULL pointer normally indicates the end of a data structure just as the null character indicates the end of a string.

Linked Lists

- A **linked list** is a linear collection of self-referential structures, called **nodes**, connected by pointer **links**—hence, the term "linked" list.
- A linked list is accessed via a pointer to the first node of the list.
- Subsequent nodes are accessed via the link pointer member stored in each node.
- By convention, the link pointer in the last node of a list is set to NULL to mark the end of the list.
- Data is stored in a linked list dynamically—each node is created as necessary.

Linked Lists

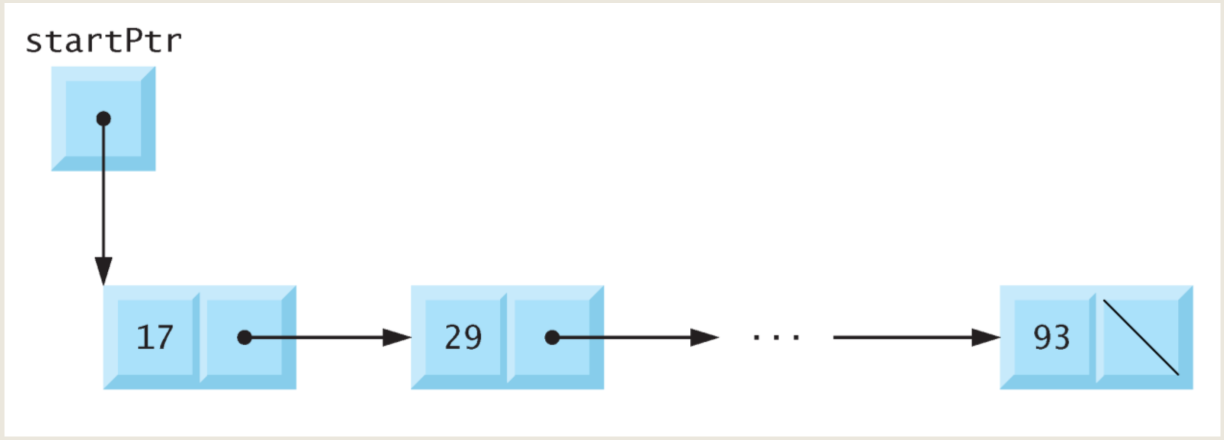
- A node can contain data of any type including other struct objects.
- Lists of data can be stored in arrays, but linked lists provide several advantages.

Linked Lists

- A linked list is appropriate when the number of data elements is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- The size of an array created at compile time, however, cannot be altered.
- Arrays can become full.
- Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

Linked Lists

- Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list.
- Linked-list nodes are normally not stored contiguously in memory.
- Logically, however, the nodes of a linked list appear to be contiguous.



Linked Lists

- Example program 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).

Linked Lists

- The primary functions of linked lists are insert and delete
- Function isEmpty is a predicate function—it does not alter the list in any way; rather it determines whether the list is empty (i.e., the pointer to the first node of the list is NULL).
- If the list is empty, 1 is returned; otherwise, 0 is returned.
- Function printList prints the list.

Linked Lists

- Characters are inserted in the list in alphabetical order.
- Function insert receives the address of the list and a character to be inserted.
- The list’s address is necessary when a value is to be inserted at the start of the list.
- Providing the address enables the list (i.e., the pointer to the first node of the list) to be modified via a call by reference.
- Because the list itself is a pointer (to its first element), passing its address creates a pointer to a pointer (i.e., double indirection).
- This is a complex notion and requires careful programming.

insert function

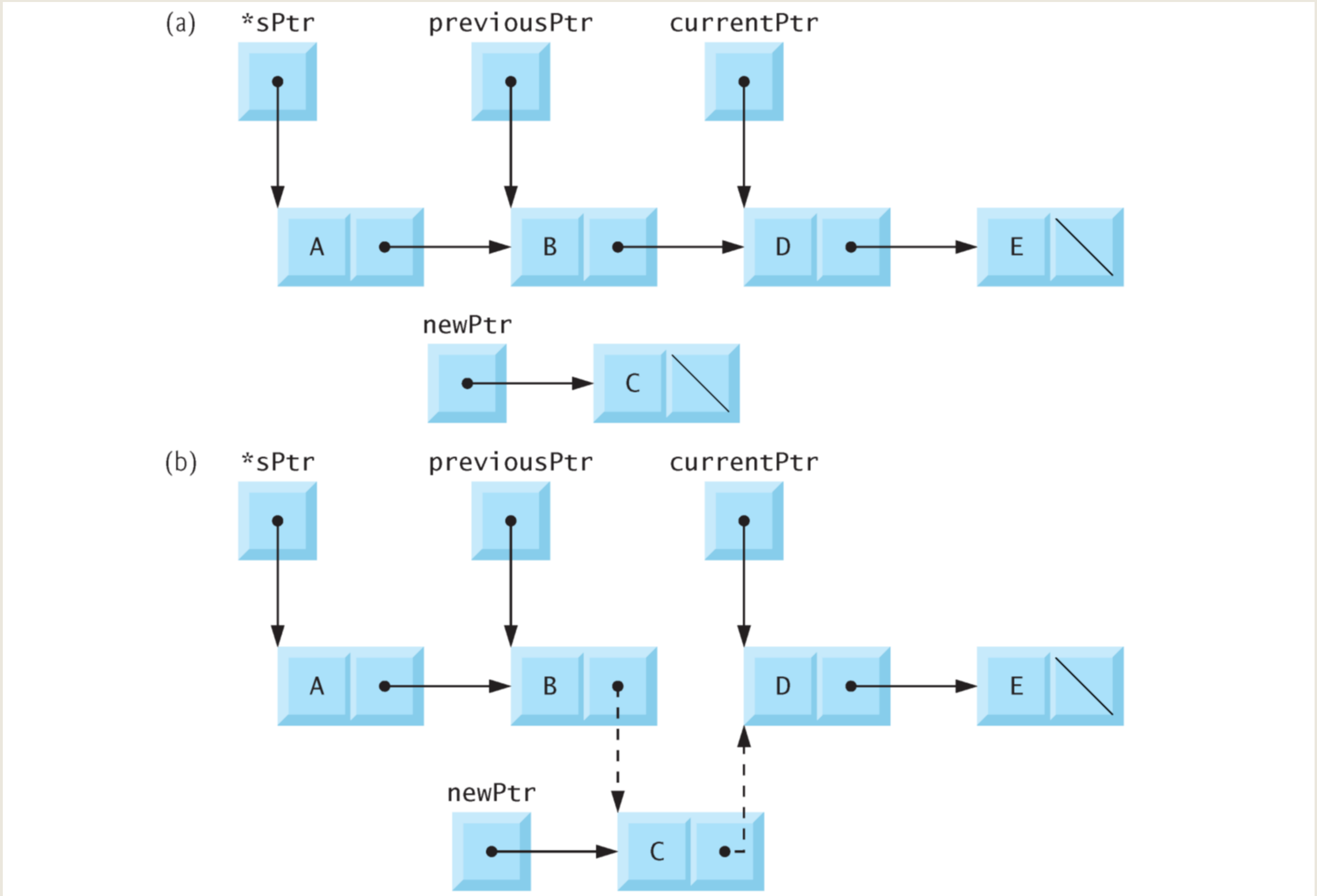
Steps for inserting a character in the list:

- Create a node: call malloc, assign to newPtr the address of the allocated memory, assign the character to be inserted to newPtr->data, and assign NULL to newPtr->nextPtr
- Initialize previousPtr to NULL and currentPtr to *sPtr—the pointer to the start of the list.
- These pointers store the locations of the node preceding the insertion point and the node after the insertion point.
- While currentPtr is not NULL and the value to be inserted is greater than currentPtr->data, assign currentPtr to previousPtr and advance currentPtr to the next node in the list
- This locates the insertion point for the value.

insert function

- If previousPtr is NULL, insert the new node as the first node in the list. Assign *sPtr to newPtr->nextPtr (the new node link points to the former first node) and assign newPtr to *sPtr (*sPtr points to the new node).
- Otherwise, if previousPtr is not NULL, the new node is inserted in place. Assign newPtr to previousPtr->nextPtr (the previous node points to the new node) and assign currentPtr to newPtr->nextPtr (the new node link points to the current node).

Inserting a node in order in a list



insert function

- Figure illustrates the insertion of a node containing the character 'C' into an ordered list.
- Part (a) of the figure shows the list and the new node just before the insertion.
- Part (b) of the figure shows the result of inserting the new node.
- The reassigned pointers are dotted arrows.
- For simplicity, we implemented function insert (and other similar functions in this chapter) with a void return type.
- It’s possible that function malloc will fail to allocate the requested memory.
- In this case, it would be better for our insert function to return a status that indicates whether the operation was successful.

delete function

Function delete receives the address of the pointer to the start of the list and a character to be deleted.

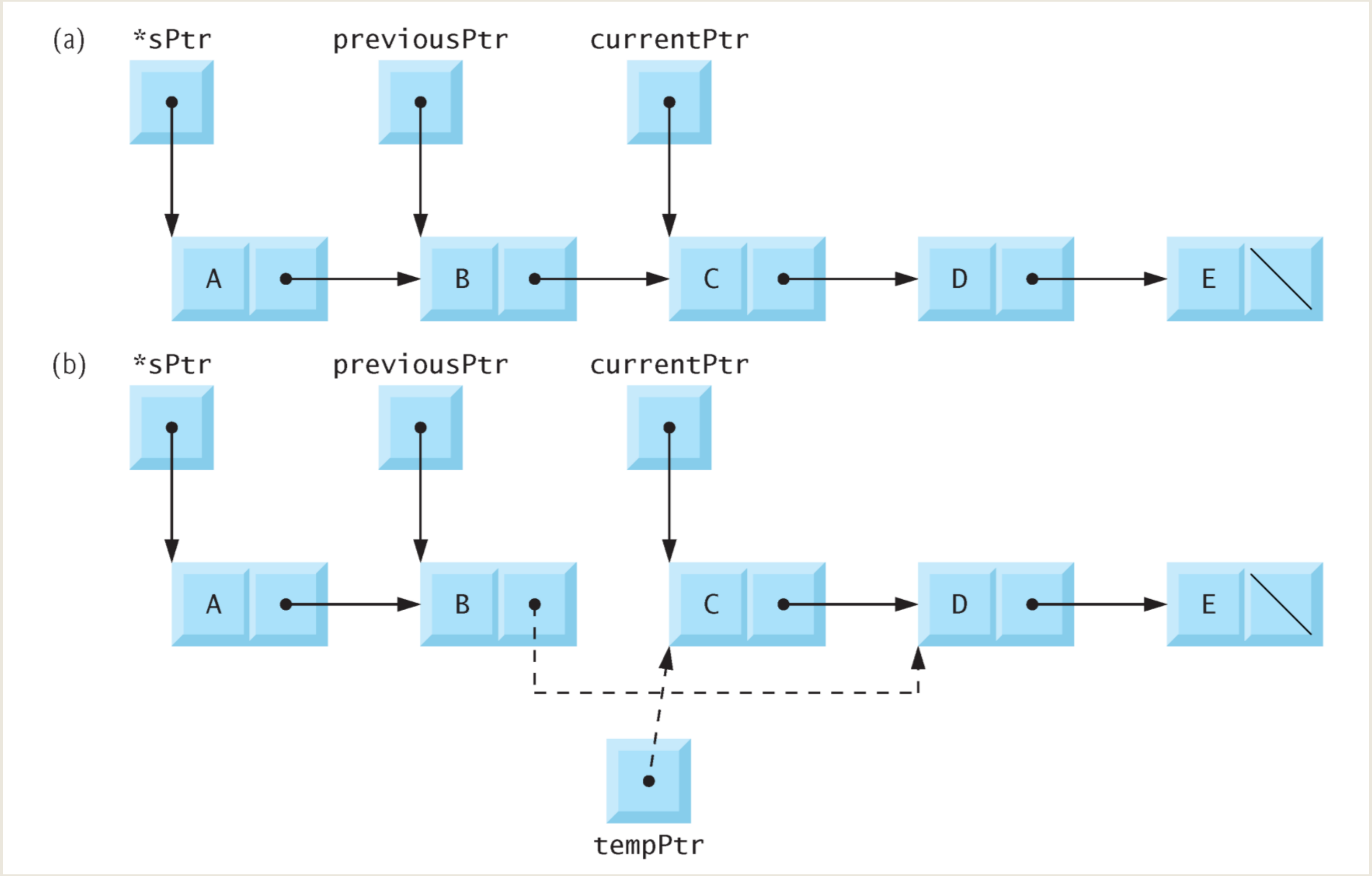
Steps for deleting a character from the list:

- If the character to be deleted matches the character in the first node of the list, assign *sPtr to tempPtr (tempPtr will be used to free the unneeded memory), assign (*sPtr)->nextPtr to *sPtr (*sPtr now points to the second node in the list), free the memory pointed to by tempPtr, and return the character that was deleted.

delete function

- Otherwise, initialize previousPtr with *sPtr and initialize currentPtr with (*sPtr)->nextPtr to advance the second node.
- While currentPtr is not NULL and the value to be deleted is not equal to currentPtr->data, assign currentPtr to previousPtr, and assign currentPtr->nextPtr to currentPtr. This locates the character to be deleted if it's contained in the list.
- If currentPtr is not NULL, assign currentPtr to tempPtr, assign currentPtr->nextPtr to previousPtr->nextPtr, free the node pointed to by tempPtr, and return the character that was deleted from the list. If currentPtr is NULL, return the null character ('\0') to signify that the character to be deleted was not found in the list.

Deleting a node from a list



delete function

- Figure illustrates the deletion of a node from a linked list.
- Part (a) of the figure shows the linked list after the preceding insert operation.
- Part (b) shows the reassignment of the link element of previousPtr and the assignment of currentPtr to tempPtr.
- Pointer tempPtr is used to free the memory allocated to the node that stores 'C'.
- Recall that we recommended setting a freed pointer to NULL.
- We do not do that in these two cases, because tempPtr is a local automatic variable and the function returns immediately.

printList function

- Function printList receives a pointer to the start of the list as an argument and refers to the pointer as currentPtr.
- The function first determines whether the list is empty and, if so, prints “List is empty." and terminates.
- Otherwise, it prints the data in the list

printList function

- While currentPtr is not NULL, the value of currentPtr->data is printed by the function, and currentPtr->nextPtr is assigned to currentPtr to advance to the next node.
- If the link in the last node of the list is not NULL, the printing algorithm will try to print past the end of the list, and an error will occur.

References

- Problem Solving and Program Design in C —7th ed, Jeri R. Hanly, Elliot B. Koffman
- C How to Program, 8ed, by Paul Deitel and Harvey Deitel, Pearson, 2016.