Lecture#8

Dynamic Memory Allocation
and
Data Structures I

CENG 102- Algorithms and Programming II,
2024-2025, Spring

Contains materials from:
P. Deitel, H. Deitel, ‘’C How to Program with an Introduction to C++’’, 8th edition, Pearson

12.1 Introduction

- So far, we have seen fixed-size data structures such as single-indexed arrays, double-indexed arrays and `struct`s.
- 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.

12.1 Introduction (Cont.)

- 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.

12.2 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`.
    ```

12.2 Self-Referential Structures (Cont.)

- 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.

12.2 Self-Referential Structures (Cont.)

- Figure 12.1 illustrates two self-referential structure objects linked together to form a list.
- A NULL pointer (represented with a diagonal line and placed in the link member of the second self-referential structure) indicates 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.

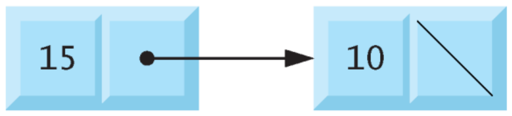


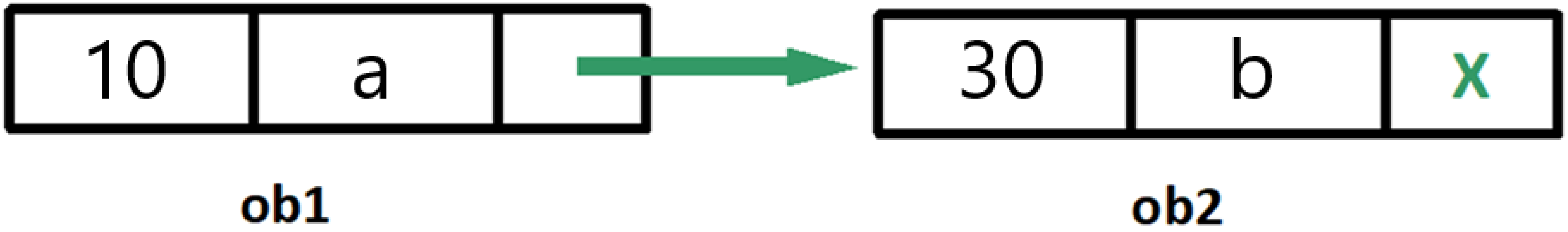
Fig. 12.1 | Self-referential structures linked together.

Common Programming Error 12.1

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

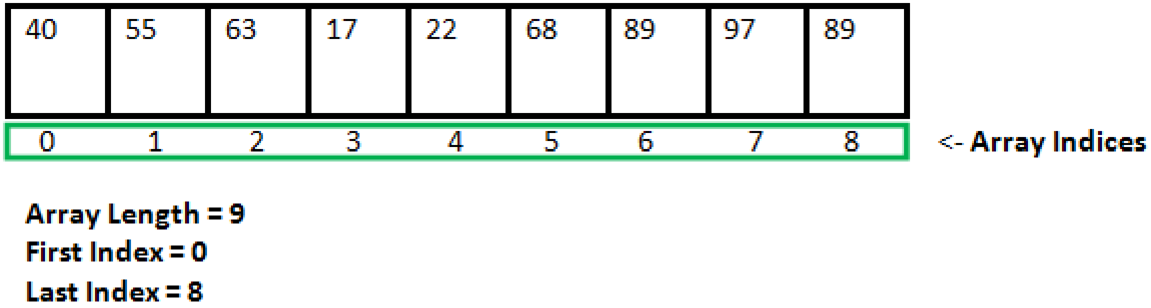
```c
#include <stdio.h>
struct node {
		int data1;
		char data2;
		struct node* link;
};
int main()
{
		struct node ob1;	// Node1
		// Initialization
		ob1.link = NULL;
		ob1.data1 = 10;
		ob1.data2 = 'a';
		struct node ob2;	// Node2
		// Initialization
		ob2.link = NULL;
		ob2.data1 = 30;
		ob2.data2 = 'b';

		// Linking ob1 and ob2
		ob1.link = &ob2;

		// Accessing data members of ob2 over ob1
		printf("%d\n", ob1.link->data1);
		printf("%c\n", ob1.link->data2);
}
```

```
30
b
```


10
a
30
b
x
ob1
ob2

12.3 Dynamic Memory Allocation

- In C language, an array is a collection of items stored at contiguous memory locations, and the size of an array is fixed, meaning that it cannot be changed at execution time.



- As can be seen, the length (size) of the array above is 9. However, what if there is a requirement to change this length (size)?
  - Positive or Negative

12.3 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.

12.3 Dynamic Memory Allocation (Cont.)

- 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.

12.3 Dynamic Memory Allocation (Cont.)

- For example, the statement

```
newPtr = malloc(sizeof(struct node));
```

  evaluates `sizeof(struct node)` to determine the size in bytes of a structure of type `struct node`, *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 (space is insufficient), malloc returns NULL.

12.3 Dynamic Memory Allocation (Cont.)

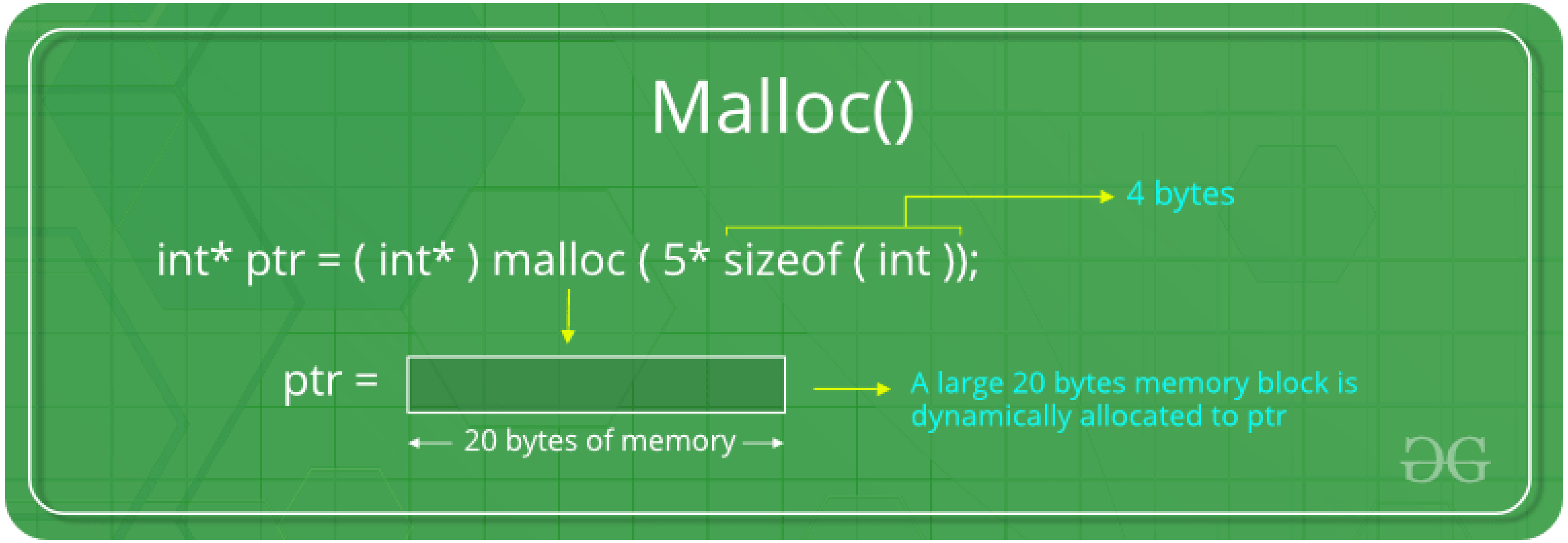
- Another example,

```
int* ptr = (int*) malloc(5 * sizeof(int));
```

- Since the size of int is 4 bytes, this statement will allocate 20 bytes of memory. And the pointer `ptr` holds the address of the first byte in the allocated memory.

- `(int*)` performs a type cast, converting a `void*` pointer returned by `malloc` into an `int*` pointer.

12.3 Dynamic Memory Allocation (Cont.)



```c
int main()
{
	int* ptr;		// This pointer will hold the base address of the block created
	int n;
	// Get the number of elements for the array
	printf("Enter number of elements:");
	scanf("%d",&n);
	// Dynamically allocate memory using malloc()
	ptr = (int*)malloc(n * sizeof(int));
	// Check if the memory has been successfully allocated by malloc or not
	if (ptr == NULL) {
		printf("Memory not allocated.\n");
		return(0);
	}
	else {
		// Memory has been successfully allocated
		printf("Memory successfully allocated using malloc.\n");
		// Get the elements of the array
		for (int i = 0; i < n; ++i) {
			ptr[i] = i + 1;
		}
		// Print the elements of the array
		printf("The elements of the array are: ");
		for (int i = 0; i < n; ++i) {
			printf("%d, ", ptr[i]);
		}
	}
}
```

Output

```
Enter number of elements: 5
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5,
```

12.3 Dynamic Memory Allocation (Cont.)

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

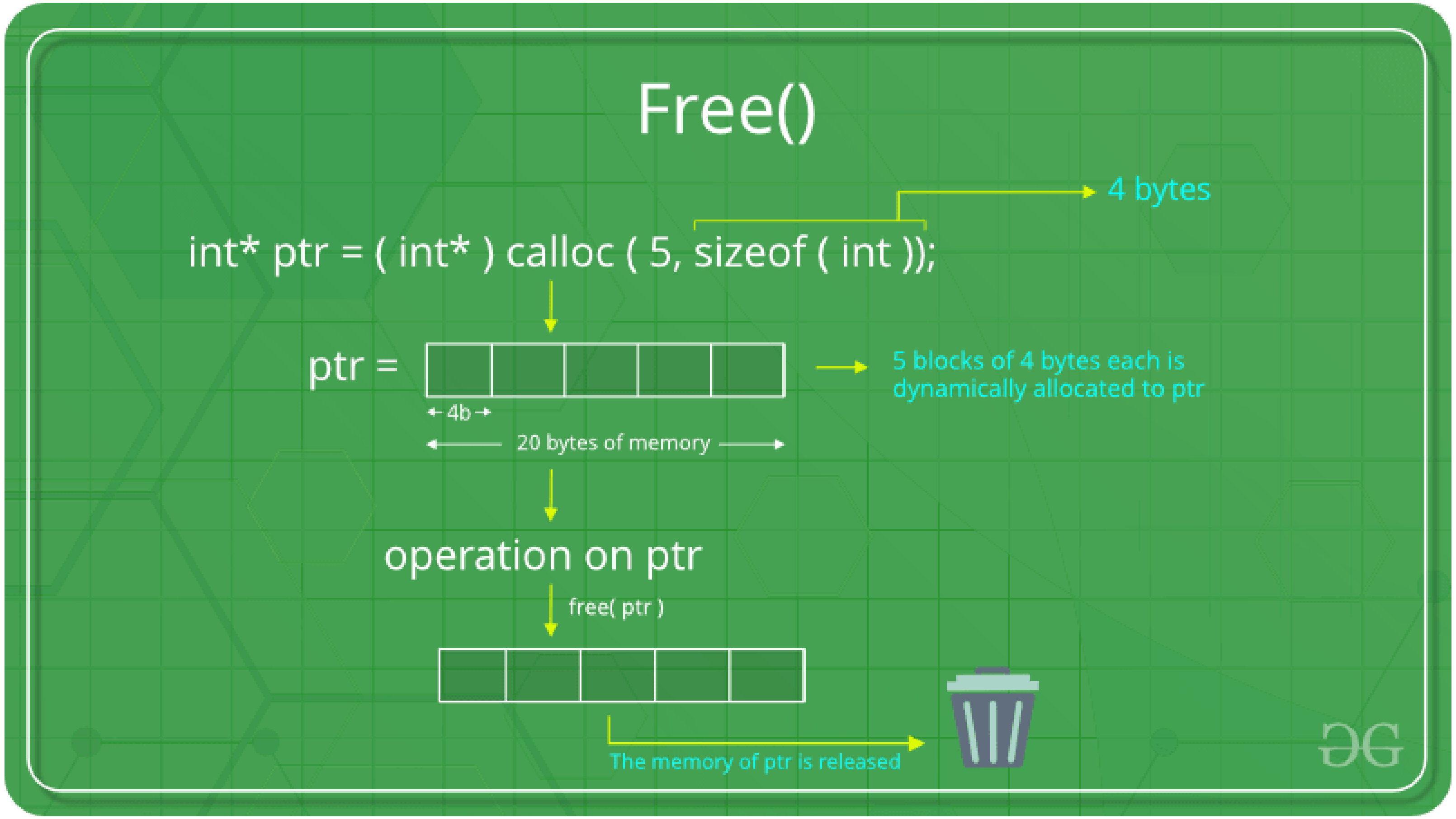
```
int* p1 = malloc(5 * sizeof(int));
int* p2 = calloc(5, sizeof(int));
p1 = (int*) realloc(p1, 10 * sizeof(int));
```

12.3 Dynamic Memory Allocation (Cont.)

- 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`, `calloc` or `realloc` call, use the statement

```
free(newPtr);
```

12.3 Dynamic Memory Allocation (Cont.)



```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int *ptrM, *ptrC;                             // These pointers will hold the base address of the blocks created
        int n = 5;                                    // The number of elements for the array
        ptrM = (int*)malloc(n * sizeof(int));         // Dynamically allocate memory using malloc()
        ptrC = (int*)calloc(n, sizeof(int));          // Dynamically allocate memory using calloc()

        // Check if the memory blocks have been successfully allocated or not
        if (ptrM == NULL || ptrC == NULL) {
                printf("Memory not allocated.\n");
                return(0);
        }
        else {
                // Memory blocks have been successfully allocated

                printf("Memory successfully allocated using malloc.\n");
                free(ptrM);                           // Free the memory
                printf("Malloc Memory successfully freed.\n");

                printf("\nMemory successfully allocated using calloc.\n");
                free(ptrC);                           // Free the memory
                printf("Calloc Memory successfully freed.\n");
        }
}
```

Portability Tip 12.1

A structure's size is not necessarily the sum of the sizes of its members.

Error-Prevention Tip 12.1

When using `malloc`*, test for a* `NULL` *pointer return value, which indicates that the memory was* not *allocated.*

Common Programming Error 12.2

Not `free`*ing dynamically allocated memory when it's no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "memory leak."*

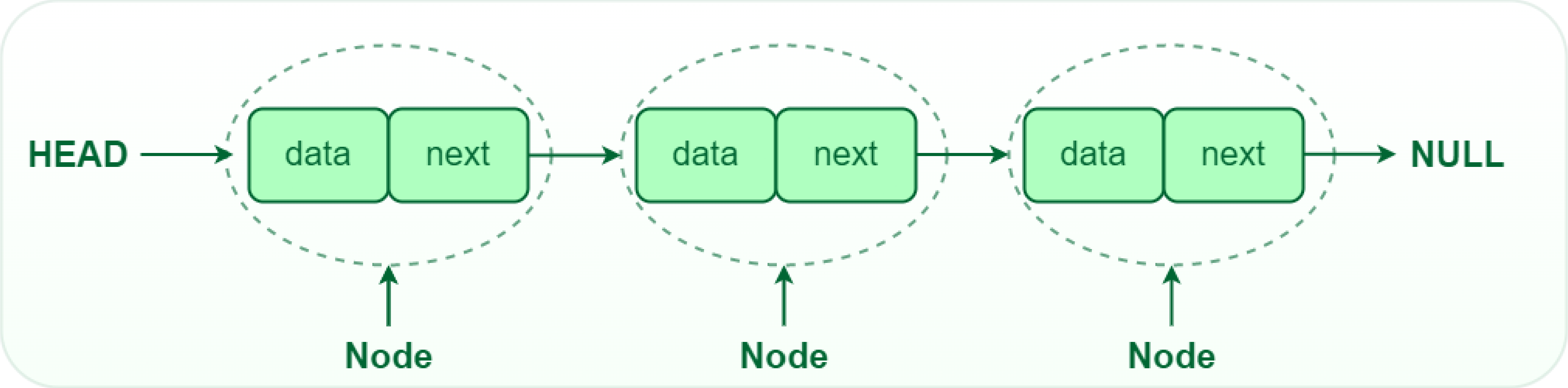
Common Programming Error 12.4

Referring to memory that has been freed is an error that typically results in the program crashing.

12.4 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.

12.4 Linked Lists



- Node Structure: A node in a linked list typically consists of two components:
  – Data: It holds the actual value or data associated with the node.
  – Next Pointer: It stores the memory address (reference) of the next node in the sequence.
- Head and Tail: The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

12.4 Linked Lists (Cont.)

- A node can contain data of *any* type including other `struct` objects.
- Stacks and queues are also linear data structures
  - Constrained versions of linked lists
- Trees are *nonlinear* data structures.

12.4 Linked Lists (Cont.)

- Lists of data can be stored in arrays, but linked lists provide several advantages.
  - A linked list is appropriate when the number of data elements is *unpredictable*.
  - Linked lists are dynamic, so the size of a list can grow or shrink, as necessary. This is not possible with arrays as the size of an array created at compile time.
  - Arrays can become full. Linked lists also can become full but only when the system has insufficient memory to satisfy dynamic storage allocation requests.
  - Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list.

Performance Tip 12.1

An array can be declared to contain more elements than the number of data items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations.

Performance Tip 12.2

Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately.

Performance Tip 12.3

The elements of an array are stored contiguously in memory. This allows immediate access to any array element because 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 to their elements.

Performance Tip 12.4

Using dynamic memory allocation (instead of arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that the pointers take up space.

12.4 Linked Lists (Cont.)

- Linked-list nodes are normally *not* stored contiguously in memory.
- Logically, however, the nodes of a linked list *appear* to be contiguous.
- Figure 12.2 illustrates a linked list with several nodes.

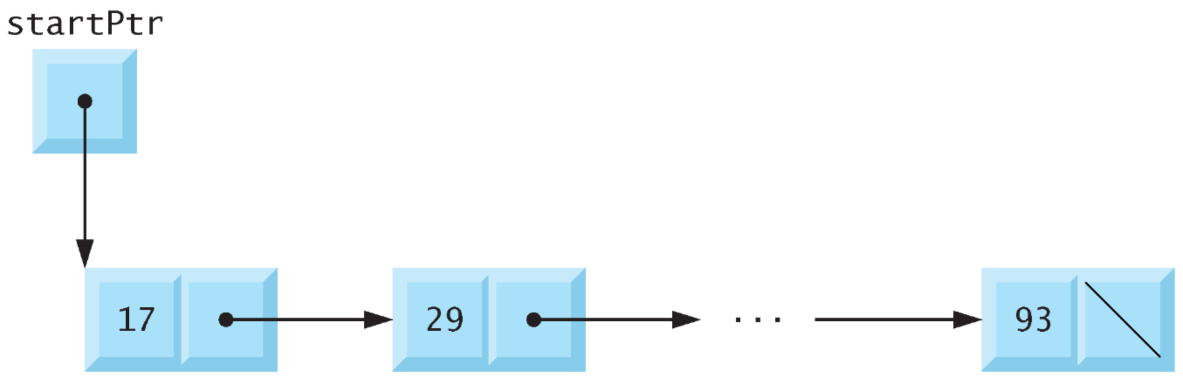


Fig. 12.2 | Linked-list graphical representation.

```c
// Linked list implementation in C
#include <stdio.h>
#include <stdlib.h>
// Creating a node
struct node {
    int value;
    struct node *next;
};
// print the linked list value
void printLinkedlist(struct node *p) {
    while (p != NULL) {
        printf("%d\n", p->value);
        p = p->next;
    }
}
int main() {
    // Initialize nodes
    struct node *head;
    struct node *one = NULL;
    struct node *two = NULL;
    struct node *three = NULL;
    // Allocate memory
    one = malloc(sizeof(struct node));
    two = malloc(sizeof(struct node));
    three = malloc(sizeof(struct node));
    // Assign value values
    one->value = 8;    // or (*one).value = 8
    two->value = 6;
    three->value = 4;
    // Connect nodes
    one->next = two;   // or (*one).next = two
    two->next = three;
    three->next = NULL;
    // printing node-value
    head = one;
    printLinkedlist(head);
}
```

12.4 Linked Lists (Cont.)

- Figure 12.3 (output shown in Fig. 12.4) 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)
{
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 1 of 8.)

```c
    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);

```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 2 of 8.)

```
                // 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);
    }

```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 3 of 8.)

```
   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;

```

Fig. 12.3 | 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);
   }
}

```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 5 of 8.)

```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
      }

```

Fig. 12.3 | 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;
}

```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 7 of 8.)

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

Fig. 12.3 | 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.
```

Fig. 12.4 | Sample output for the program of Fig. 12.3. (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.
```

Fig. 12.4 | Sample output for the program of Fig. 12.3. (Part 2 of 2.)

12.4 Linked Lists (Cont.)

- 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.

12.4 Linked Lists (Cont.)

- 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.

12.4 Linked Lists (Cont.)

- Steps for inserting a character in the list (Fig. 12.5):
  - *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.

12.4 Linked Lists (Cont.)

- 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).

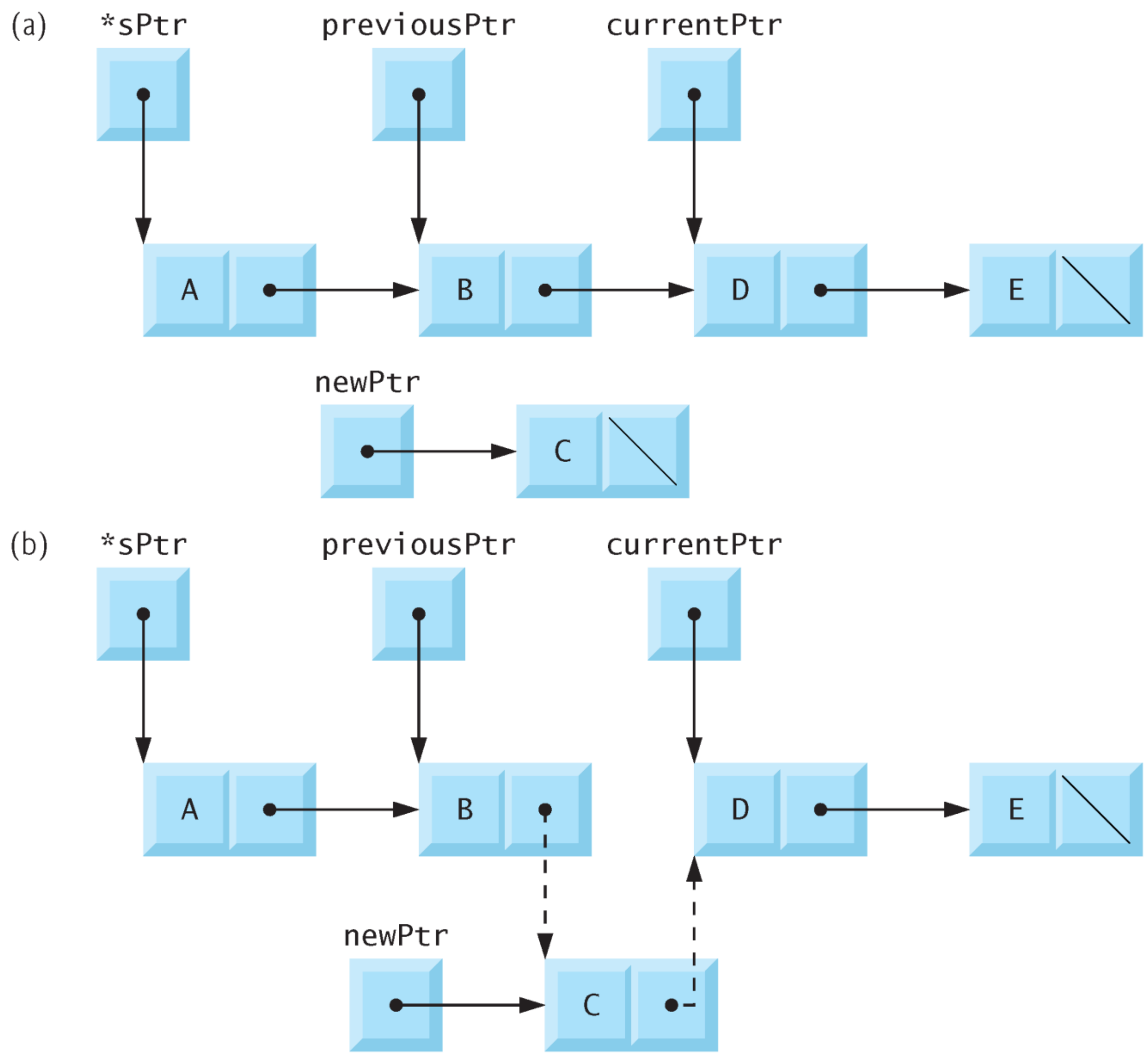


Fig. 12.5 | Inserting a node in order in a list.

12.4 Linked Lists (Cont.)

- Figure 12.5 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.

12.4.2 Function `delete`

- 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.
  - 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.

12.4.2 Function `delete` (Cont.)

– 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.

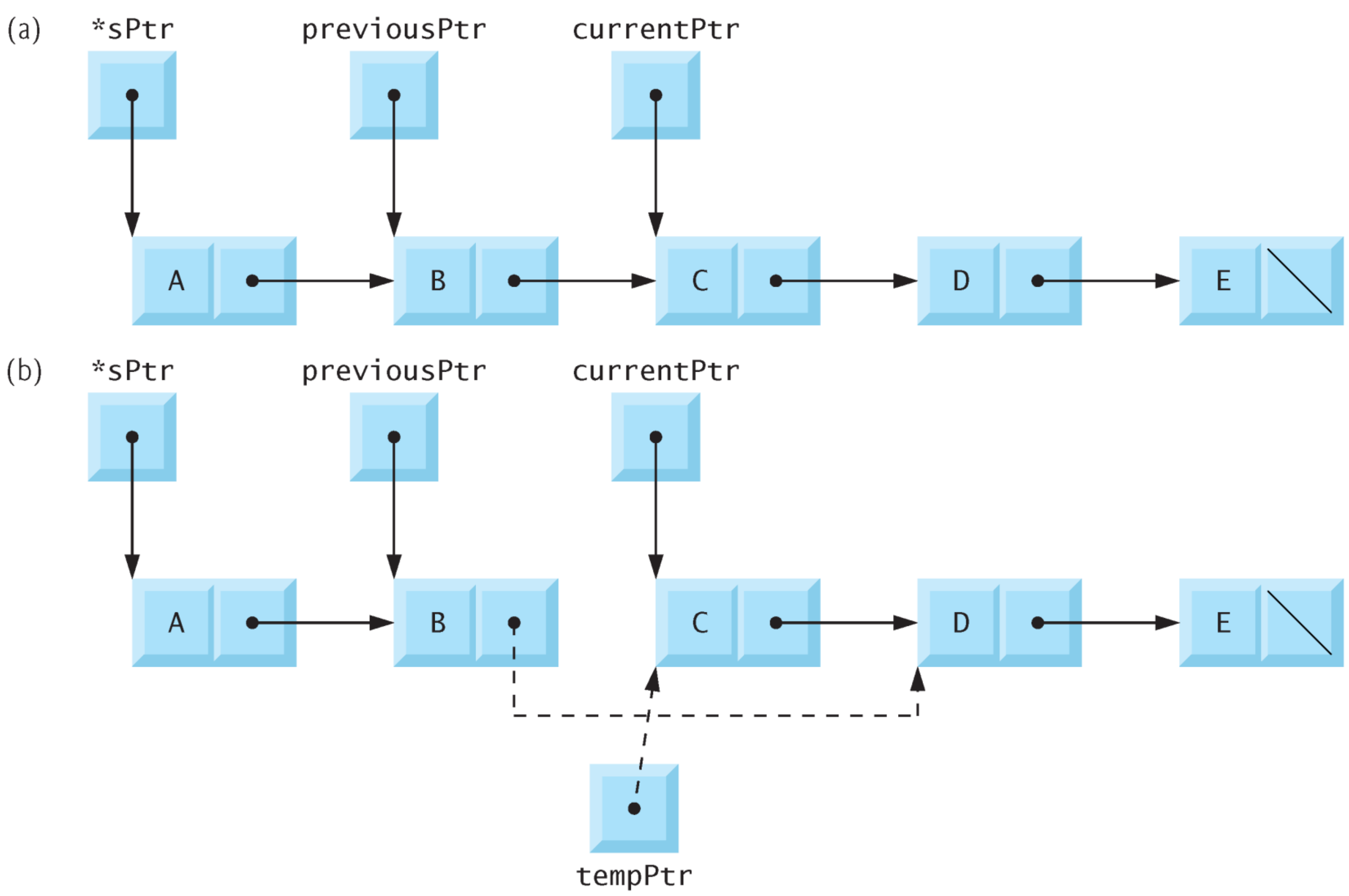


Fig. 12.6 | Deleting a node from a list.

12.4.2 Function `delete` (Cont.)

- Fig. 12.6 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.

12.4.3 Function `printList`

- 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

12.4 Linked Lists (Cont.)

- 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.
- The printing algorithm is identical for linked lists, stacks and queues.