Chapter 4
Structures

Specific Learning Outcomes

The students must have the ability to:

1. Understand the use of structures and how it is processed;
2. Pass structures to functions by value and by reference;
3. Allocate and free memory dynamically for data objects;
4. Form linked data structures using pointers, self-referential structures and recursion;
5. Create and manipulate linked lists;
6. Understand various important applications of linked data structures.

Learning Content

1. Uses of Structures;
2. Declaring and Defining Structures;
3. Declaring and Defining Nested Structures;
4. Passing Structures to a Function;
5. Self-Referential Structures;
6. Dynamic Memory Allocation;
7. Linked Lists.

Structures

Structures—sometimes referred to as aggregates in the C standard—are collections of related variables under one name. Structures may contain variables of many different data types—in contrast to arrays, which contain only elements of the same data type. Structures are commonly used to define records to be stored in files. Pointers and structures facilitate the formation of more complex data structures such as linked lists, queues, stacks and trees. We will also discuss typedefs for creating aliases for previously defined data types.

Structures are derived data types—they're constructed using objects of other types. Consider the following structure definition:

```
struct card {
char *face;
char *suit;
};
```

Keyword struct introduces a structure definition. The identifier card is the structure tag, which names the structure definition and is used with struct to declare variables of the structure type—e.g., struct card. Variables declared within the braces of the structure definition are the structure's members. Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict. Each structure definition must end with a semicolon.

Common Programming Error. Forgetting the semicolon that terminates a structure definition is a syntax error.

The definition of struct card contains members face and suit, each of type char *. Structure members can be variables of the primitive data types (e.g., int, float, etc.), or aggregates, such as arrays and other structures. As we saw in Arrays, each element of an array must be of the same type. Structure members, however, can be of different types. For example, the following struct contains character array members for an employee's first and last names, an unsigned int member for the employee's age, a char member that would contain 'M' or 'F' for the employee's gender and a double member for the employee's hourly salary:

```
struct employee {
  char firstName[20];
  char lastName[20];
  unsigned int age;
  char gender;
  double hourlySalary;
};
```

Self-Referential Structures

A variable of a struct type cannot be declared in the definition of that same struct type. A pointer to that struct type, however, may be included. For example, in struct employee2:

```
struct employee2 {
  char firstName[20];
  char lastName[20];
  unsigned int age;
  char gender;
  double hourlySalary;
  struct employee2 teamLeader; // ERROR
  struct employee2 *teamLeaderPtr; // pointer
};
```

the instance of itself (teamLeader) is an error. Because teamLeaderPtr is a pointer (to type struct employee2), it's permitted in the definition. A structure containing a member that's a pointer to the same structure type is referred to as a self-referential structure. Self-referential structures are used to build linked data structures.

Common Programming Error. A structure cannot contain an instance of itself.

Defining Variables of Structure Types

Structure definitions do not reserve any space in memory; rather, each definition creates a new data type that's used to define variables—like a blueprint of how to build instances of that struct. Structure variables are defined like variables of other types. The definition

```
struct card aCard, deck[52], *cardPtr;
```

declares aCard to be a variable of type struct card, declares deck to be an array with 52 elements of type struct card and declares cardPtr to be a pointer to struct card. After the preceding statement, we've reserved memory for one struct card object named aCard, 52 struct card objects in the deck array and an uninitialized pointer of type struct card. Variables of a given structure type may also be declared by placing a comma separated list of the variable names between the closing brace of the structure definition and the semicolon that ends the structure definition. For example, the preceding definition could have been incorporated into the struct card definition as follows:

```
struct card {
  char *face;
  char *suit;
} aCard, deck[52], *cardPtr;
```

Structure Tag Names

The structure tag name is optional. If a structure definition does not contain a structure tag name, variables of the structure type may be declared only in the structure definition - not in a separate declaration.

Common Programming Error. Always provide a structure tag name when creating a structure type. The structure tag name is required for declaring new variables of the structure type later in the program.

Operations That Can Be Performed on Structures

The only valid operations that may be performed on structures are:

- assigning struct variables to struct variables of the same type.
- taking the address (&) of a struct variable.
- accessing the members of a struct variable.
- using the sizeof operator to determine the size of a struct variable.

Common Programming Error. Assigning a structure of one type to a structure of a different type is a compilation error.

Structures may not be compared using operators == and !=, because structure members are not necessarily stored in consecutive bytes of memory. Sometimes there are "holes" in a structure, because computers may store specific data types only on certain memory boundaries such as half-word, word or double-word boundaries. A word is a memory unit used to store data in a computer—usually 4 bytes or 8 bytes. Consider the following structure definition, in which sample1 and sample2 of type struct example are declared:

```
struct example {
  char c;
  int i;
} sample1, sample2;
```

A computer with 4-byte words might require that each member of struct example be aligned on a word boundary, i.e., at the beginning of a word—this is machine dependent. Figure 4.1 shows a sample storage alignment for a variable of type struct example that has been assigned the character 'a' and the integer 97 (the bit representations of the values are shown). If the members are stored beginning at word boundaries, there's a three-byte hole (bytes 1–3 in the figure) in the storage for variables of type struct example. The value in the three-byte hole is undefined. Even if the member values of sample1 and sample2 are in fact equal, the structures are not necessarily equal, because the undefined three-byte holes are not likely to contain identical values.

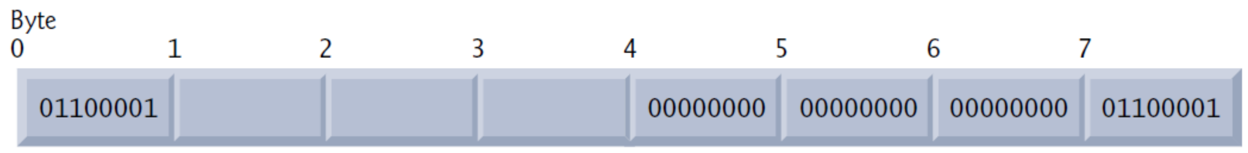

Figure 4.1

Possible storage alignment for a variable of type struct example showing an undefined area in memory

Because the size of data items of a particular type is machine dependent and because storage alignment considerations are machine dependent, so too is the representation of a structure.

Initializing Structures

Structures can be initialized using initializer lists as with arrays. To initialize a structure, follow the variable name in the definition with an equals sign and a brace-enclosed, comma-separated list of initializers. For example, the declaration

```
struct card aCard = { "Three", "Hearts" };
```

creates variable aCard to be of type struct card and initializes member face to "Three" and member suit to "Hearts". If there are fewer initializers in the list than members in the structure, the remaining members are automatically initialized to 0 (or NULL if the member is a pointer). Structure variables defined outside a function definition (i.e., externally) are initialized to 0 or NULL if they're not explicitly initialized in the external definition. Structure variables may also be initialized in assignment statements by assigning a structure variable of the same type, or by assigning values to the individual members of the structure.

Accessing Structure Members with . and ->

Two operators are used to access members of structures: the structure member operator (.)—also called the dot operator—and the structure pointer operator (->)—also called the arrow operator. The structure member operator accesses a structure member via the structure variable name. For example, to print member suit of structure variable aCard defined in Section 10.3, use the statement

```
printf("%s", aCard.suit); // displays Hearts
```

The structure pointer operator—consisting of a minus (-) sign and a greater than (>) sign with no intervening spaces—accesses a structure member via a pointer to the structure. Assume that the pointer cardPtr has been declared to point to struct card and that the address of structure aCard has been assigned to cardPtr.

To print member suit of structure aCard with pointer cardPtr, use the statement

printf("%s", cardPtr->suit); // displays Hearts

The expression cardPtr->suit is equivalent to (*cardPtr).suit, which dereferences the pointer and accesses the member suit using the structure member operator. The parentheses are needed here because the structure member operator (.) has a higher precedence than the pointer dereferencing operator (*). The structure pointer operator and structure member operator, along with parentheses (for calling functions) and brackets ([]) used for array indexing, have the highest operator precedence and associate from left to right.

Good Programming Practice. Do not put spaces around the -> and . operators. Omitting spaces helps emphasize that the expressions the operators are contained in are essentially single variable names.

Good Programming Practice. Inserting space between the - and > components of the structure pointer operator or between the components of any other multiple-keystroke operator except ?: is a syntax error.

Good Programming Practice. Attempting to refer to a structure member by using only the member's name is a syntax error.

Good Programming Practice. Not using parentheses when referring to a structure member that uses a pointer and the structure member operator (e.g., *cardPtr.suit) is a syntax error. To prevent this problem use the arrow (->) operator instead.

Program 32 demonstrates the use of the structure member and structure pointer operators. Using the structure member operator, the members of structure aCard are assigned the values "Ace" and "Spades", respectively (lines 17 and 18). Pointer cardPtr is assigned the address of structure aCard (line 20). Function printf prints the members of structure variable aCard using the structure member operator with variable name aCard, the structure pointer operator with pointer cardPtr and the structure member operator with dereferenced pointer cardPtr (lines 22–24).

```
1 // Program_32.c
2 // Structure member operator and
3 // structure pointer operator
4 #include <stdio.h>
5
6 // card structure definition
7 struct card {
8         char *face; // define pointer face
9         char *suit; // define pointer suit
10 };
```

```

int main(void)
{
	struct card aCard; // define one struct card variable

	// place strings into aCard
	aCard.face = "Ace";
	aCard.suit = "Spades";

	struct card *cardPtr = &aCard; // assign address of aCard to cardPtr

	printf("%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
		cardPtr->face, " of ", cardPtr->suit,
		(*cardPtr).face, " of ", (*cardPtr).suit);
}
```

output

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Program 32 Structure member operator and structure pointer operator

Using Structures with Functions

Structures may be passed to functions by

- passing individual structure members.
- passing an entire structure.
- passing a pointer to a structure.

When structures or individual structure members are passed to a function, they're passed by value. Therefore, the members of a caller's structure cannot be modified by the called function. To pass a structure by reference, pass the address of the structure variable. Arrays of structures - like all other arrays - are automatically passed by reference. In Arrays, we stated that you can use a structure to pass an array by value. To do so, create a structure with the array as a member. Structures are passed by value, so the array is passed by value.

Common Programming Error. Assuming that structures, like arrays, are automatically passed by reference and trying to modify the caller's structure values in the called function is a logic error.

Passing structures by reference is more efficient than passing structures by value (which requires the entire structure to be copied).

Typedef

The keyword typedef provides a mechanism for creating synonyms (or aliases) for reviously defined data types. Names for structure types are often defined with typedef to create shorter type names. For example, the statement

```
typedef struct card Card;
```

defines the new type name Card as a synonym for type struct card. C programmers often use typedef to define a structure type, so a structure tag is not required. For example, the following definition

```
typedef struct {
  char *face;
  char *suit;
} Card;
```

creates the structure type Card without the need for a separate typedef statement.

Good Programming Practice. Capitalize the first letter of typedef names to emphasize that they're synonyms for other type names.

Card can now be used to declare variables of type struct card. The declaration

```
Card deck[52];
```

declares an array of 52 Card structures (i.e., variables of type struct card). Creating a new name with typedef does not create a new type; typedef simply creates a new type name, which may be used as an alias for an existing type name. A meaningful name helps make the program self-documenting. For example, when we read the previous declaration, we know "deck is an array of 52 Cards."

Often, typedef is used to create synonyms for the basic data types. For example, a program requiring four-byte integers may use type int on one system and type long on another. Programs designed for portability often use typedef to create an alias for fourbyte integers, such as Integer. The alias Integer can be changed once in the program to make the program work on both systems. Use typedef to help make a program more portable.

Good Programming Practice. Using typedefs can help make a program more readable and maintainable.

We have studied fixed-size data structures such as one-dimensional arrays, two-dimensional arrays and structs. We will now introduces dynamic data structures that can grow and shrink at execution time.

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. 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” (i.e., link) 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. Figure 4.2 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. [Note: The slash is only for illustration purposes; it does not correspond to the backslash character in C.] A NULL pointer normally indicates the end of a data structure just as the null character indicates the end of a string.

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

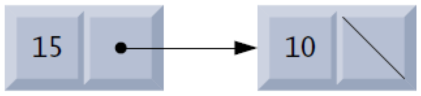

Figure 4.2

Self-referential structures linked together

Dynamic Memory Allocation

Creating and maintaining dynamic data structures that can grow and shrink as the program runs 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. 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. For example, the statement

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

evaluates sizeof(struct node) to determine a struct node object's size in bytes, allocates a new area in memory of that number of bytes and stores a pointer to the allocated memory in newPtr. The memory is not guaranteed to be initialized, though many implementations initialize it for security. If no memory is available, malloc returns NULL. 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. These functions are discussed in advance sections. The sections that follow discuss lists is created and maintained with dynamic memory allocation and self-referential structures. A structure's size is not necessarily the sum of the sizes of its members. This is so because of various machine-dependent boundary alignment requirements.

Error-Prevention Tip. When using malloc, test for a NULL pointer return value, which indicates that the memory was not allocated.

Common Programming Error. Not freeing 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."

Error-Prevention Tip. When memory that was dynamically allocated is no longer needed, use free to return the memory to the system immediately. Then set the pointer to NULL to eliminate the possibility that the program could refer to memory that's been reclaimed and which may have already been allocated for another purpose.

Common Programming Error. Freeing memory not allocated dynamically with malloc is an error.

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

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. A node can contain data of any type including other structs. Stacks and queues are also linear data structures, and, as we’ll see, are constrained versions of linked lists. Trees are nonlinear data structures.

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 to be represented in the data structure is unpredictable. Linked lists are dynamic, so the length of a list can increase or decrease at execution time 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.

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. Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list. Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately. The elements of an array are stored contiguously in memory. This allows immediate accessto 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.

Linked-list nodes are normally not stored contiguously in memory. Logically, however, the nodes of a linked list appear to be contiguous. Figure 4.3 illustrates a linked list with several nodes. 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, and that dynamic memory allocation incurs the overhead of function calls.

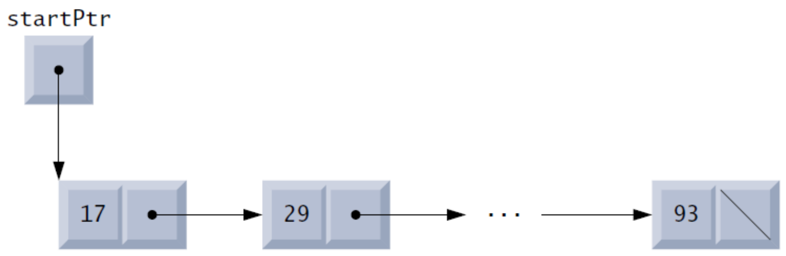

Figure 4.3

Linked-list graphical representation

Program 33 manipulates a list of characters. You can insert a character in the list in alphabetical order (function insert) or delete a character from the list (function delete). A detailed discussion of the program follows.

```
// Program_33.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)
{
    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);

                    // 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);
        }

        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;

                // 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);
        }
}

// 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
                }

                // 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;
}

// 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");
        }
}
```

outpu

```
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.
? 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.
```

Program 33 Inserting and deleting nodes in a list

The primary functions of linked lists are insert (lines 85–115) and delete (lines 118– 147). Function isEmpty (lines 150–153) is called 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 (lines 156–173) prints the list.

Function insert

Characters are inserted in the list in alphabetical order. Function insert (lines 85–115) 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. The steps for inserting a character in the list are as follows (see Figure 4.4):

1. Create a node by calling malloc, assigning to newPtr the address of the allocated memory (line 87), assigning the character to be inserted to newPtr->data (line 90), and assigning NULL to newPtr->nextPtr (line 91).
2. Initialize previousPtr to NULL (line 93) and currentPtr to *sPtr (line 94)—the pointer to the start of the list. Pointers previousPtr and currentPtr store the locations of the node preceding and after the insertion point, respectively.
3. While currentPtr is not NULL and the value to be inserted is greater than currentPtr-> data (line 97), assign currentPtr to previousPtr (line 98) and advance currentPtr to the next node in the list (line 99). This locates the insertion point for the value.
4. If previousPtr is NULL (line 103), insert the new node as the first in the list (lines 104–105). Assign *sPtr to newPtr->nextPtr (the new node link points to the for-mer first node) and assign newPtr to *sPtr (*sPtr points to the new node). Otherwise, if previousPtr is not NULL, insert the new node in place (lines 108–109). 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).

Error-Prevention Tip. Assign NULL to a new node's link member. Pointers should be initialized before they're used.

Figure 4.4 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.

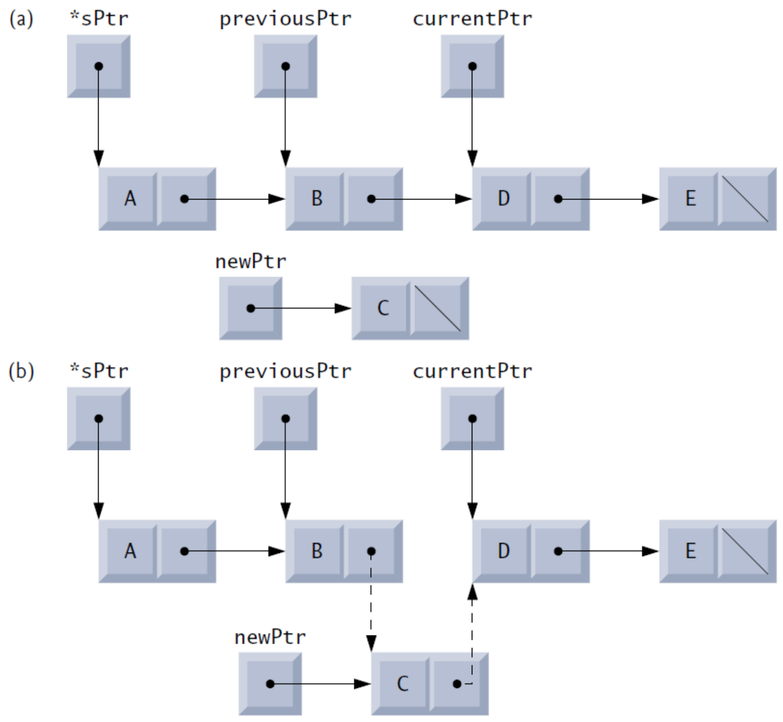

Figure 4.4

Inserting a node in order in a list

Function delete

Function delete (lines 118–147) receives the address of the pointer to the start of the list and a character to be deleted. The steps for deleting a character from the list are as follows (see Figure 4.5):

1. If the character to be deleted matches the character in the first node of the list (line 121), 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.
2. Otherwise, initialize previousPtr with *sPtr and initialize currentPtr with (*sPtr)->nextPtr (lines 128–129) to advance to the second node.
3. While currentPtr is not NULL and the value to be deleted is not equal to currentPtr-> data (line 132), assign currentPtr to previousPtr (line 133) and assign currentPtr->nextPtr to currentPtr (line 134). This locates the character to be deleted if it's contained in the list.
4. If currentPtr is not NULL (line 138), assign currentPtr to tempPtr (line 139), assign currentPtr->nextPtr to previousPtr->nextPtr (line 140), free the node pointed to by tempPtr (line 141), and return the character that was deleted from the list (line 142). If currentPtr is NULL, return the null character ('\0') to signify that the character to be deleted was not found in the list (line 146).

Figure 4.5 illustrates the deletion of the node containing the character 'C' 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'. Note that in lines 124 and 141 we free tempPtr. 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.

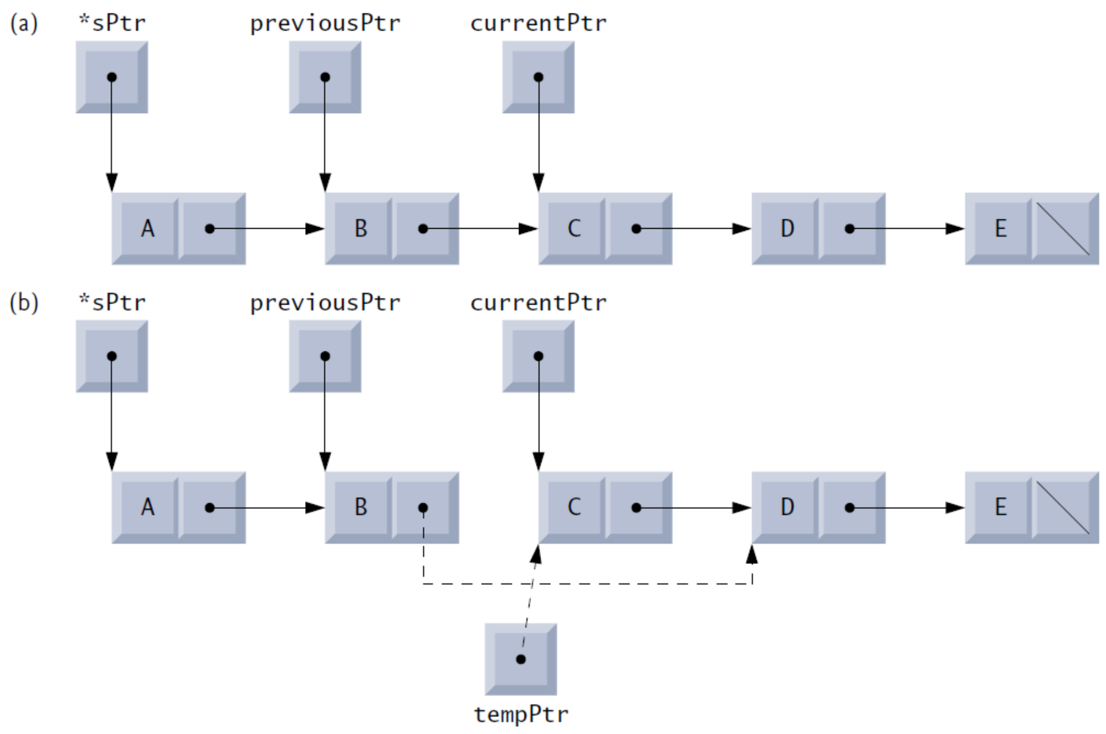

Figure 4.5

Deleting a node from a list

Function printList

Function printList (lines 156–173) 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 (lines 159–161) and, if so, prints "List is empty." and terminates. Otherwise, it prints the data in the list (lines 162–172). 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.

References

Paul Deitel, and Harvey Deitel, “C How to Program with an introduction to C++” Eight Edition

Kimberly Nelson King, “C Programming: A Modern Approach” Second Edition

Brian W. Kernighan and Dennis M. Ritchie, “C Programming Language” 2nd Edition

Jeri R. Hanly and Elliot B. Koffman, “Problem Solving and Program Design in C”, 8th Edition

https://www.thecrazyprogrammer.com

https://www.geeksforgeeks.org

Additional Resources and Links

Introduction to Structures in C by Neso Academy https://www.youtube.com/ watch?v=zmRxC7gYw- &list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=150

Declaring Structure Variables by Neso academy https://www.youtube.com/ watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=151

Structure Types (Using Structure Tags) by Neso Academy https://www.youtube.com/ watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=152

Structure Types (Using typedef) by Neso Academy https://www.youtube.com/ watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=153

Initializing & Accessing the Structure Members by Neso Academy https://www.youtube.com/ watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubu F5QextdCSM&index=154

Designated Initialization in Structures by Neso Academy https://www.youtube.com/watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=155

Declaring an Array of Structure by Neso Academy https://www.youtube.com/watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=156

Pointer to Structure Variable by Neso Academy https://www.youtube.com / watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=157

Structures in C (Solved Problem 1) by Neso Academy https://www.youtube.com/watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=160

Structures in C (Solved Problem 2) by Neso Academy https://www.youtube.com/watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=161

Structures in C (Solved Problem 3) by Neso Academy https://www.youtube.com/watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=162

Program to Find Area of Rectangle Using Structures by Neso Academy https://www.youtube.com/watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=168

Basics of Dynamic Memory Allocation by Neso Academy https://www.youtube.com/watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=188

Dynamic Memory Allocation using malloc() by Neso Academy https://www.youtube.com/ watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=188

Releasing the Dynamically Allocated Memory using free() by Neso Academy https://www.youtube.com/watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=188

Self Referential Structures by Neso Academy https://www.youtube.com/watch?v=otu7gJVcwDw

Introduction to Linked List by Neso Academy https://www.youtube.com/watch?v=b5QR4AmrspU

Array vs. Single Linked List (In Terms of Representation) by Neso Academy https://www.youtube.com/watch?v=zmRxC7gYw-g&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=188

Creating the Node of a Single Linked Lis by Neso Academy
https://youtu.be/Vch7_YeGKH4

Creating a Single Linked List (Part 1) by Neso Academy
https://youtu.be/nxtDe6Gq4t4

Creating a Single Linked List (Part 2) by Neso Academy
https://youtu.be/HrY_YmU1vdg

Chapter Assessment

Answer the following.

I. Fill in the blanks in each of the following:

a. A(n) _____ is a collection of related variables under one name.
b. The variables declared in a structure definition are called its _____.
c. Keyword _____ introduces a structure declaration.
d. Keyword _____ is used to create a synonym for a previously defined data type.
e. The name of the structure is referred to as the structure _____.
f. A structure member is accessed with either _____ the or the _____ operator.
g. A self- _____ structure is used to form dynamic data structures.
h. The pointer to the next node in a linked list is referred to as a(n) _____.

II. State whether each of the following is true or false. If false, explain why.

a. Structures may contain variables of only one data type.
b. The tag name of a structure is optional.
c. Members of different structures must have unique names.
d. Keyword typedef is used to define new data types.
e. Structures are always passed to functions by reference.
f. Structures may not be compared by using operators == and !=.

III. Write code to accomplish each of the following:

a. Define a structure called part containing unsigned int variable partNumber and char array partName with values that may be as long as 25 characters (including the terminating null character).
b. Define Part to be a synonym for the type struct part.
c. Use Part to declare variable a to be of type struct part, array b[10] to be of type struct part and variable ptr to be of type pointer to struct part.
d. Read a part number and a part name from the keyboard into the individual members of variable a.
e. Assign the member values of variable a to element 3 of array b.
f. Assign the address of array b to the pointer variable ptr.
g. Print the member values of element 3 of array b using the variable ptr and the structure pointer operator to refer to the members.
h. Write the definition for a self-referential structure for a binary node Bnode that will be an element of a binary tree,i.e a node that has an integer data and two pointers to same type of node corresponding to the left and right child.

i. Declare a pointer root, that will point to the root node of the binary tree defined in #h but is initially NULL.
j. Create the root node for the binary tree with the following info

Notes

Notes

Notes