CMPE 252
C PROGRAMMING

SPRING 2021

WEEK 14-15

POINTERS AND DYNAMIC DATA STRUCTURES
CHAPTER 13

Problem Solving & Program Design in C

Eighth Edition
Global Edition

Jeri R. Hanly & Elliot B. Koffman

Chapter Objectives

- To understand dynamic allocation on the heap
- To learn how to use pointers to access structs
- To learn how to use pointers to build linked data structures
- To understand how to use and implement a linked list
- To understand how to use and implement a stack
- To learn how to use and implement a queue
- To understand basic concepts of binary trees
- To learn how to use and implement a binary tree

Terminology

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

TABLE 13.1 Pointer Uses Already Studied

Use	Implementation
Function output parameters	1. Function formal parameter declared as a pointer type. 2. Actual parameter in a call is the address of a variable.
Arrays (strings)	1. Declaration of array variable shows array size. 2. Name of array with no subscript is a pointer: It means the address of initial array element.
File access	1. Variable declared of type `FILE *` is a pointer to a structure that is to contain access information for a file. 2. File I/O functions such as `fscanf`, `fprintf`, `fread`, and `fwrite` expect as arguments file pointers of type `FILE *`.
Function as a parameter of another function	1. Declaration may or may not include a *. 2. Name of a function alone (with no parameter list) is a pointer to the function's code.

Function Pointers

- A function pointer is a variable that stores the address of a function that can later be called through that function pointer.
- This is useful because functions encapsulate behavior.
  - For instance, every time you need a particular behavior such as drawing a line, instead of writing out a bunch of code, all you need to do is call the function.
  - But sometimes you would like to choose different behaviors at different times in essentially the same piece of code.

Source: https://www.cprogramming.com/tutorial/function-pointers.html

Function Pointers Syntax

- void (*foo)(int);

- In this example, foo is a pointer to a function taking one argument, an integer, and that returns void.
- It's as if you're declaring a function called "*foo", which takes an int and returns void;
  - now, if *foo is a function, then foo must be a pointer to a function. (Similarly, a declaration like int *x can be read as *x is an int, so x must be a pointer to an int.)

Function Pointers

- A pointer to a function

```c
#include <stdio.h>
void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    // Invoking fun() using fun_ptr
    (*fun_ptr)(10);

    return 0;
}
```

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

Value of a is 10

:the ampersand is actually optional
(functions already point to that piece of code)

Reading Function Pointer Declarations

- E.g. : void *(*foo)(int *);
- interpret this as:
  - (*foo) is a function which returns a void* and takes int* as argument.
  - foo is a pointer to just such a function.

Another example

```c
#include <stdio.h>

void foo (int i)
{
  printf ("foo %d!\n", i);
}
void bar (int i)
{
  printf ("%d bar!\n", i);
}
void message (void (*func)(int), int times)
{
  int j;
  for (j=0; j<times; ++j)
    func (j);  /* (*func) (j); would be equivalent. */
}

int main ()
{
    int want_foo=1;

  void (*pf)(int) = &bar; /* The & is optional. */
  if (want_foo)
    pf = foo;
  message (pf, 5);

  return;
}
```

```
foo 0!
foo 1!
foo 2!
foo 3!
foo 4!
```

Dynamic Memory Allocation

- heap
  - region of memory in which function malloc dynamically allocates blocks of storage
- stack
  - region of memory in which function data areas are allocated and reclaimed
- Both in RAM

Stack vs Heap

Stack:

•Variables created on the stack will go out of scope and are automatically deallocated.

•Much faster to allocate in comparison to variables on the heap.

•Implemented with an actual stack data structure.

•Stores local data, return addresses, used for parameter passing.

•Can have a stack overflow when too much of the stack is used (mostly from infinite or too deep recursion, very large allocations).

•Data created on the stack can be used without pointers.

•You would use the stack if you know exactly how much data you need to allocate before compile time and it is not too big.

•Usually has a maximum size already determined when your program starts. [3]

Stack vs Heap

Heap:

•In C++, variables on the heap must be destroyed manually and never fall out of scope. The data is freed with `free`.

•Slower to allocate in comparison to variables on the stack.

•Used on demand to allocate a block of data for use by the program.

•Can have fragmentation when there are a lot of allocations and deallocations.

•In C++ or C, data created on the heap will be pointed to by pointers and allocated with `new` or `malloc` respectively.

•Can have allocation failures if too big of a buffer is requested to be allocated.

•You would use the heap if you don't know exactly how much data you will need at run time or if you need to allocate a lot of data.

•Responsible for memory leaks.

[3]

```c
int* aPtr = (int*)malloc(sizeof(int)*10);
int aArr[10];
printf("Size of aPtr is: %2d\n",sizeof(aPtr));
printf("Size of aArr is: %d\n",sizeof(aArr));

int* temp = aPtr;
printf("aPtr is: %X\n", aPtr);
printf("temp is: %X\n", temp);

for(int i = 0; i < 10; i++)
    *(aPtr+i) = i+1;
for(int i = 0; i < 10; i++)
    printf("%d ",*(aPtr+i));
puts("");


//free(aPtr);
aPtr = (int*)malloc(sizeof(int)*10);
for(int i = 0; i < 10; i++)
    *(aPtr+i) = 2*(i+1);
for(int i = 0; i < 10; i++)
    printf("%d ",*(aPtr+i));
puts("");
printf("aPtr is: %X\n", aPtr);
printf("temp is: %X\n", temp);

free(aPtr);
```

```
Size of aPtr is:  4
Size of aArr is: 40
aPtr is: 1E0D60
temp is: 1E0D60
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
aPtr is: 1E0D90
temp is: 1E0D60
```

```c
int* aPtr = (int*)malloc(sizeof(int)*10);
int aArr[10];
printf("Size of aPtr is: %2d\n",sizeof(aPtr));
printf("Size of aArr is: %d\n",sizeof(aArr));

int* temp = aPtr;
printf("aPtr is: %X\n", aPtr);
printf("temp is: %X\n", temp);

for(int i = 0; i < 10; i++)
    *(aPtr+i) = i+1;
for(int i = 0; i < 10; i++)
    printf("%d ",*(aPtr+i));
puts("");


free(aPtr);
aPtr = (int*)malloc(sizeof(int)*10);
for(int i = 0; i < 10; i++)
    *(aPtr+i) = 2*(i+1);
for(int i = 0; i < 10; i++)
    printf("%d ",*(aPtr+i));
puts("");
printf("aPtr is: %X\n", aPtr);
printf("temp is: %X\n", temp);

free(aPtr);
```

```
Size of aPtr is:  4
Size of aArr is: 40
aPtr is: 2D70D60
temp is: 2D70D60
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
aPtr is: 2D70D60
temp is: 2D70D60
```

no malloc after free

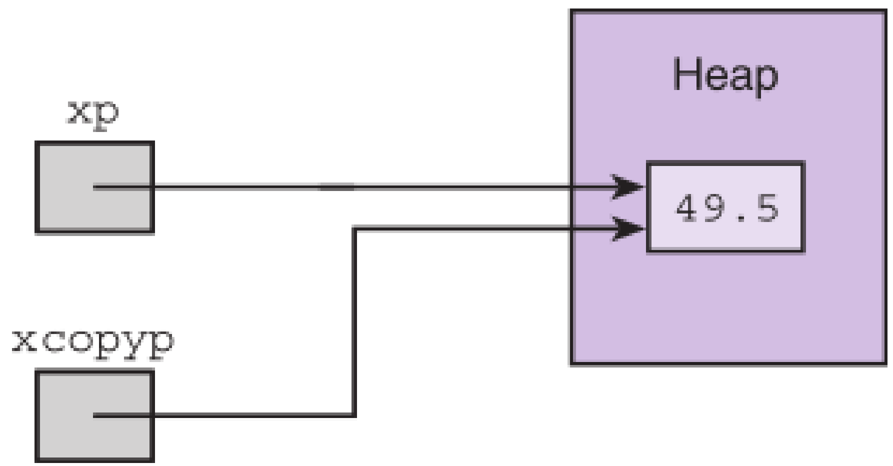
```
free(aPtr);

 printf("aPtr is: %X\n", aPtr);
 for(int i = 0; i < 10; i++)
    printf("%d ",*(aPtr+i));
```

undefined behaviour

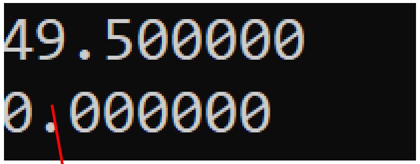
```
temp is: 2A60D60
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
aPtr is: 2A60D60
temp is: 2A60D60
aPtr is: 2A60D60
44447208 44436800 6 8 10 12 14 16 18 20
```

FIGURE 13.9

Multiple Pointers to a Cell in the Heap




```c
double *xp, *xcopy;
xp = (double*) malloc (sizeof(double));
*xp = 49.5;
xcopy = xp;
printf("%f\n",*xcopy);
free(xp);
printf("%f\n",*xcopy);
```



This is garbage – not actual 0
freeing doesnot zeroize that memory bloc
Just returns that location back to heap

FIGURE 13.9

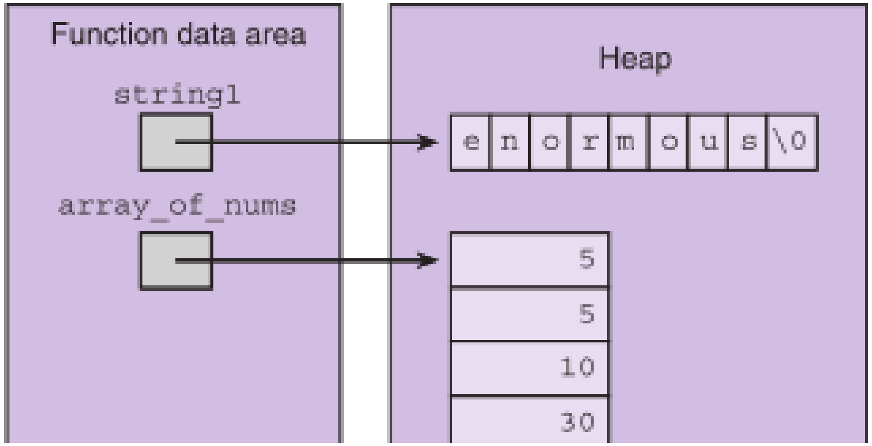
Multiple Pointers to a Cell in the Heap

```c
int *xp,*xcopy;
xp =(int*) malloc(sizeof(int));
*xp=49;
xcopy=xp;
printf("%d\n",*xcopy);
free(xp);
printf("%d\n",*xcopy);
printf("%d\n",*xp);
```

```
49
11013496
11013496
```

Garbage after freeing

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


int main()
{
	char	*string1;
	int	*array_of_nums;

	int	 str_siz, num_nums;
	printf("Enter string length and string> ");
	scanf("%d", &str_siz);
                                      str_siz+1
	string1 = (char *)calloc(str_siz, sizeof (char));
	scanf("%s", string1);

	printf("\nHow many numbers?> ");
	scanf("%d", &num_nums);

	array_of_nums = (int *)calloc(num_nums, sizeof (int));
	array_of_nums[0] = 5;

	for (int i = 1; i < num_nums; ++i)
		array_of_nums[i] = array_of_nums[i - 1] * i;

}
```



Function data area
string1
array_of_nums
Heap
e n o r m o u s \0
5
5
10
30

Linked Lists

Linked Lists

- linked list
  - a sequence of nodes in which each node but the last contains the address of the next node
- empty list
  - a list of no nodes
  - represented in C by the pointer NULL, whose value is zero
- list head
  - the first element in a linked list

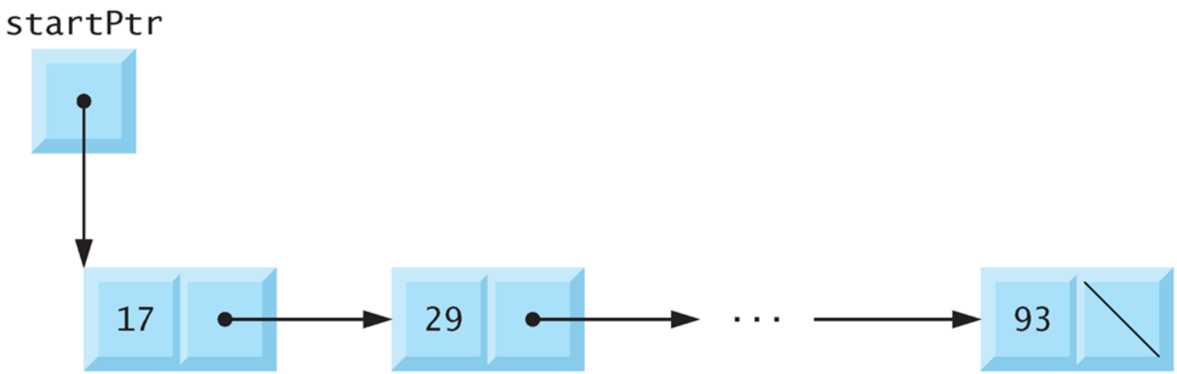


Fig. 12.2 | Linked-list graphical representation.

in all nodes but the last, link pointer contains the address of the next node

we do not know how many elements we have, dynamic allocation is required

If you create a node struct, how should it look like?

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

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.
- Stacks and queues are also linear data structures
  - 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 is *unpredictable*.

Linked Lists

- 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.

Example from book

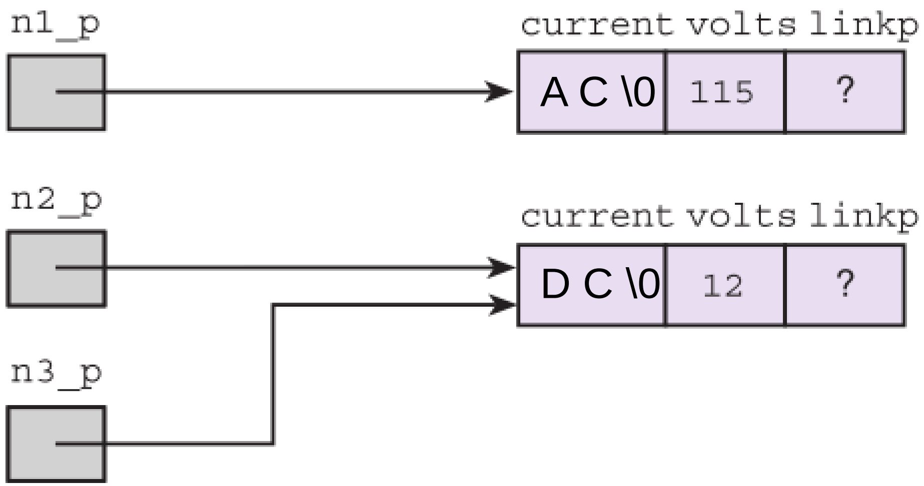


FIGURE 13.11

Multiple Pointers to the Same Structure

```
typedef struct node_s {
      char              current[3];
      int               volts;
      struct node_s *linkp;
} node_t;
```

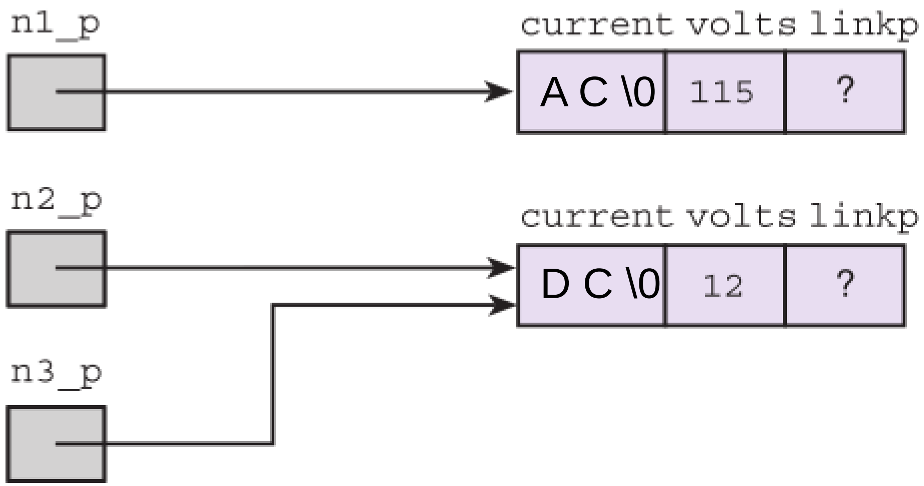
We can allocate and initialize the data components of two nodes as follows:

```
node_t *n1_p, *n2_p, *n3_p;
n1_p = (node_t *)malloc(sizeof (node_t));
strcpy(n1_p->current, "AC");
n1_p->volts = 115;
n2_p = (node_t *)malloc(sizeof (node_t));
strcpy(n2_p->current, "DC");
n2_p->volts = 12;
```

If we then copy the pointer value of n2_p into n3_p ,

```
n3_p = n2_p;
```

See the figure



We can compare two pointer expressions using the equality operators == and != .

The following conditions are all true for our node_t * variables n1_p , n2_p , and n3_p .

n1_p != n2_p   n1_p != n3_p   n2_p == n3_p

Connecting Nodes

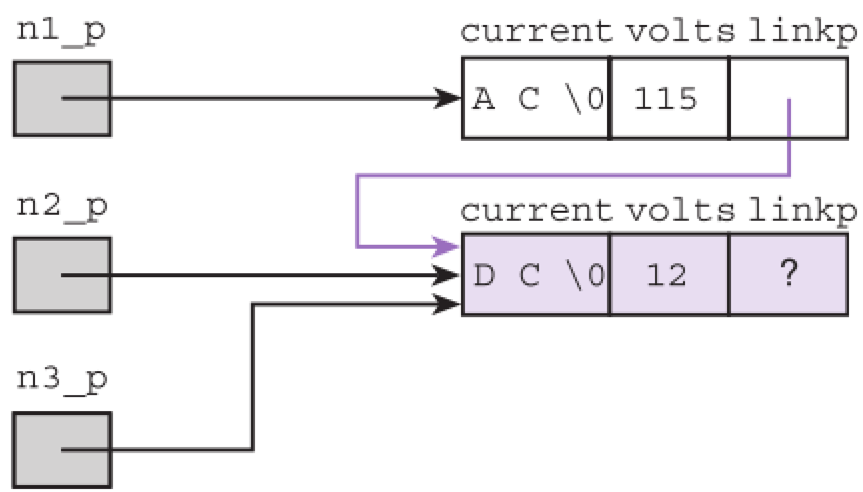
One purpose of using dynamically allocated nodes is to enable us to grow data structures of varying size

We accomplish this by connecting individual nodes.

n1_p->linkp = n2_p;

FIGURE 13.12

Linking Two Nodes



We now have *three* ways to access the 12 in the volts component of the second node:

n2_p->volts / n3_p->volts / n1_p->linkp->volts

n1_p->volts=n1_p->linkp->volts;

(*n1_p).volts=(*(*n1_p).linkp).volts

TABLE 13.2 Analyzing the Reference n1_p->linkp->volts

Section of Reference	Meaning
`n1_p->linkp`	Follow the pointer in `n1_p` to a structure and select the `linkp` component.
`linkp->volts`	Follow the pointer in the `linkp` component to another structure and select the `volts` component.

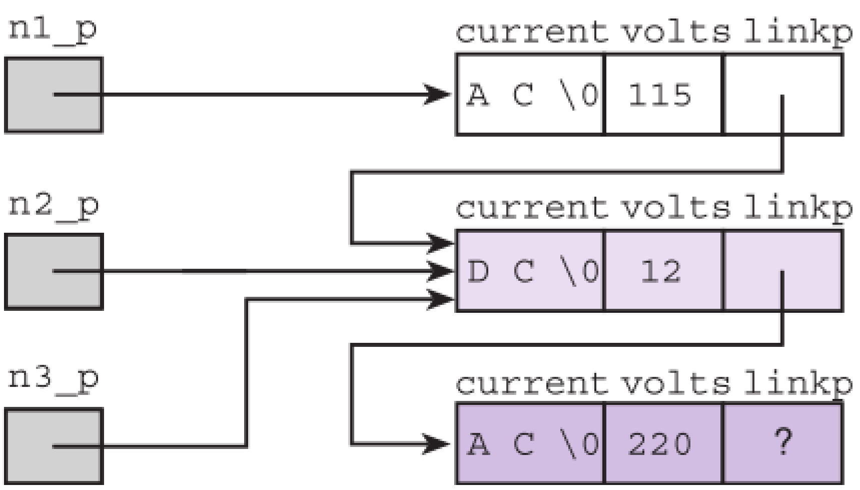
The linkp component of our structure with three access paths is still undefined, so we will allocate a third node, storing its pointer in this link.

```
n2_p->linkp = (node_t *)malloc(sizeof (node_t));
strcpy(n2_p->linkp->current, "AC");
n2_p->linkp->volts = 220;
```

Now we have the three-node linked list shown in Fig. 13.13

FIGURE 13.13

Three-Node Linked List with Undefined Final Pointer

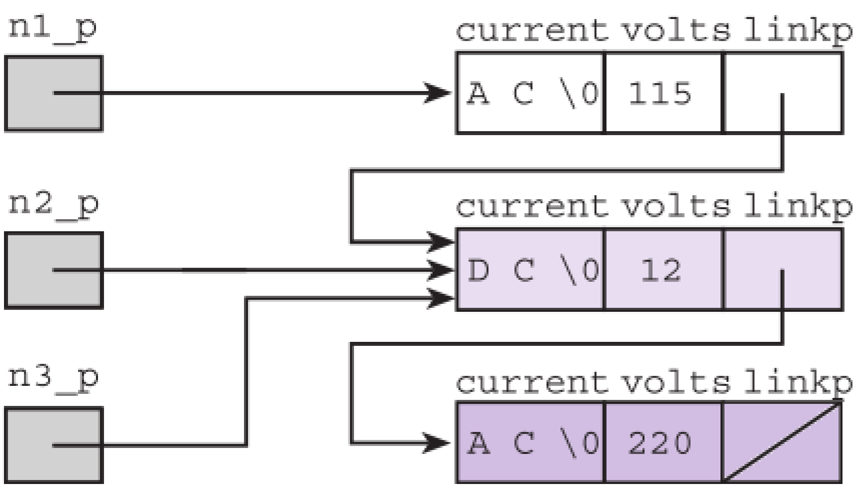


!However, we still have an undefined linkp component at the end

At some point our list must end, and we need a special value to mark the end showing that the linked list of nodes following the current node is empty.

In C, the **empty list** is represented by the pointer NULL , which we will show in our memory diagrams as a diagonal line through a pointer variable or component.

Execute : n2_p->linkp->linkp = NULL;



in Fig. 13.14 , we have a complete linked list whose length is three.

The pointer variable n1_p points to the first list element, or **list head**.

Any function that knows this address in n1_p would have the ability to access every element of the list.

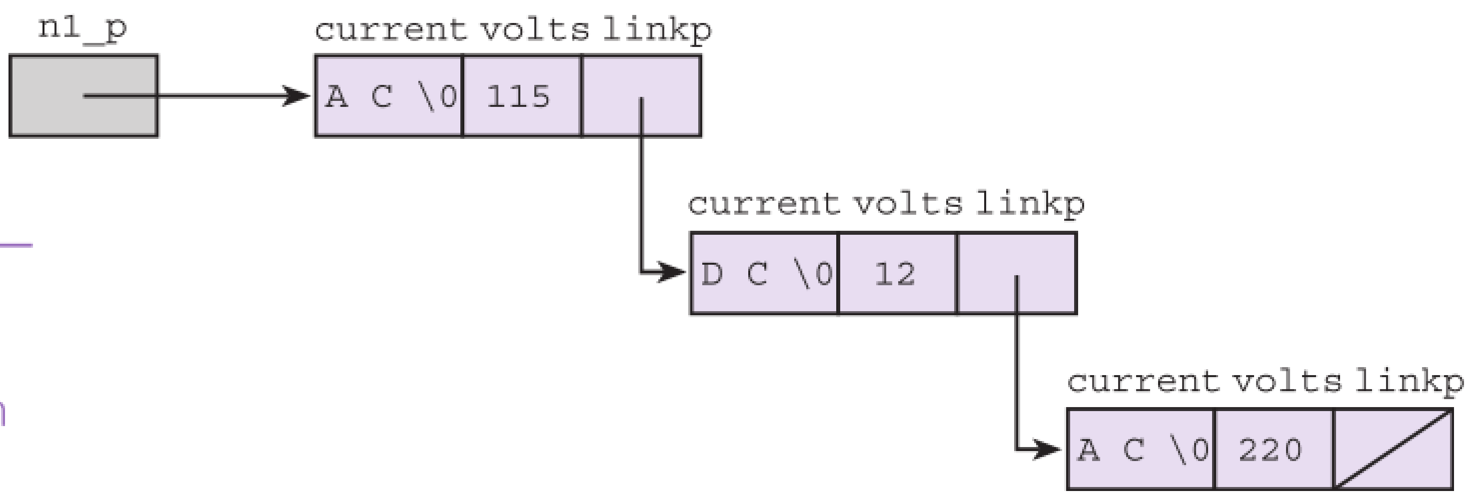


FIGURE 13.14

Three-Element Linked List Accessed Through n1_p

Advantages of Linked Lists

- It can be modified easily.
- The means of modifying a linked list works regardless of how many elements are in the list.
- It is easy to delete an element.

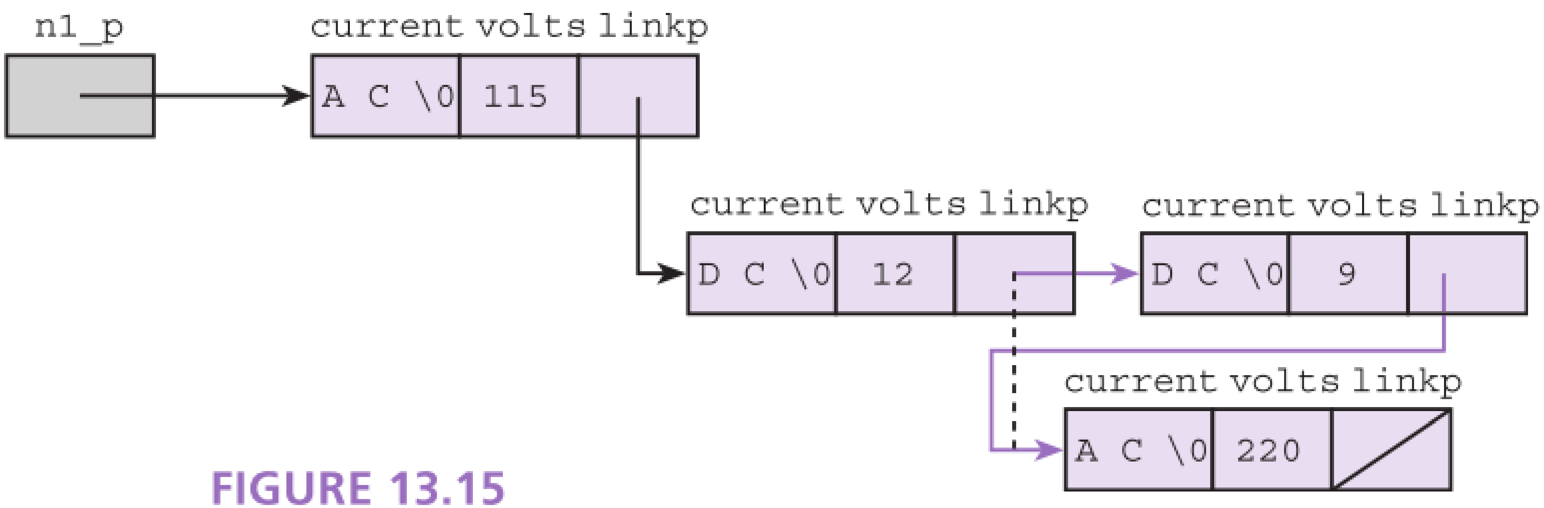
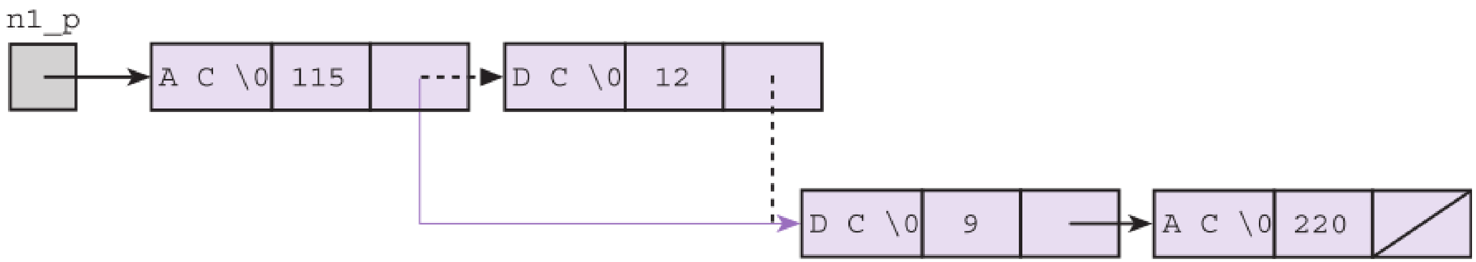


FIGURE 13.15

Linked List After an Insertion

```
node_t *p = (node_t *) malloc(sizeof(node_t));
strcpy(p->current,"DC");
p->volts = 9;
p->linkp = n1_p->linkp->linkp;
n1_p->linkp->linkp = p;
```

FIGURE 13.16 Linked List After a Deletion




```
node_t *p = n1_p->linkp;
n1_p->linkp = n1_p->linkp->linkp;
free(p);
```


Linked List Operators

- traversing a list
  - processing each node in a linked list in sequence, starting at the list head
- tail recursion
  - any recursive call that is executed as a function's last step

```
typedef struct list_node_s {
      int                    digit;
      struct list_node_s *restp;
} list_node_t;
. . .
{
      list_node_t *pi_fracp;
```

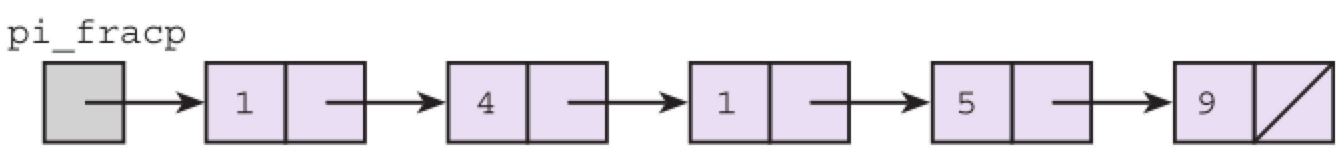


FIGURE 13.17 Function print_list

```
/*
 * Displays the list pointed to by headp
 */
void
print_list(list_node_t *headp)
{
      if (headp == NULL) {    /* simple case - an empty list                    */
            printf("\n");
      } else {                /* recursive step - handles first element         */
            printf("%d", headp->digit);  /*       leaves rest to                */
            print_list(headp->restp);    /*       recursion                     */
      }
}
```


FIGURE 13.18 Comparison of Recursive and Iterative List Printing

```
/* Displays the list pointed to by headp */
void
print_list(list_node_t *headp)
```

```
{
 if (headp == NULL) {/* simple case */
     printf("\n");
 } else {           /* recursive step */
     printf("%d", headp->digit);
     print_list(headp->restp);
 }
}
```

```
{ list_node_t *cur_nodep;

  for  (cur_nodep = headp; /* start at
                              beginning */
        cur_nodep != NULL; /* not at
                              end yet   */
        cur_nodep = cur_nodep->restp)
      printf("%d", cur_nodep->digit);
  printf("\n");
}
```

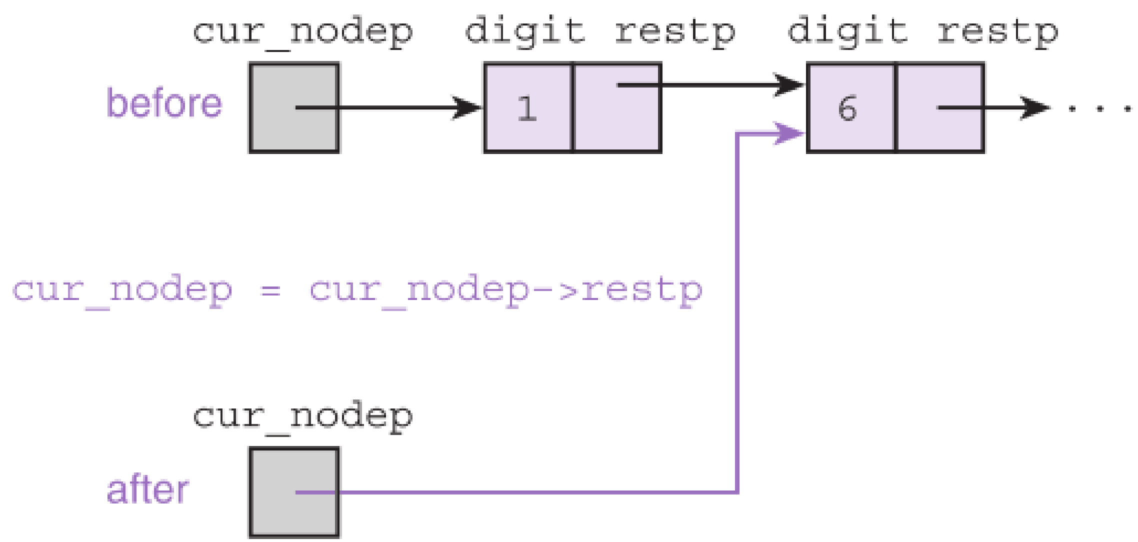



FIGURE 13.19

Update of List-Traversing Loop Control Variable

FIGURE 13.20 Recursive Function get_list

```
#include <stdlib.h>  /*  gives access to malloc */
#define SENT -1
/*
 *  Forms a linked list of an input list of integers
 *  terminated by SENT
 */
list_node_t *
get_list(void)
{
     int data;
     list_node_t *ansp;

     scanf("%d", &data);
     if (data == SENT) {
           ansp = NULL;
     } else {
           ansp = (list_node_t *)malloc(sizeof (list_node_t));
           ansp->digit = data;
           ansp->restp = get_list();
     }

     return (ansp);
}
```

FIGURE 13.21 Iterative Function get_list

```
/*
 *  Forms a linked list of an input list of integers terminated by SENT
 */
list_node_t *
get_list(void)
{
      int data;
      list_node_t *ansp,
                  *to_fillp, /* pointer to last node in list whose
                                restp component is unfilled          */
                  *newp;     /* pointer to newly allocated node      */

      /* Builds first node, if there is one */
      scanf("%d", &data);
      if (data == SENT) {
            ansp = NULL;
      } else {
            ansp = (list_node_t *)malloc(sizeof (list_node_t));
            ansp->digit = data;
            to_fillp = ansp;

            /* Continues building list by creating a node on each
               iteration and storing its pointer in the restp component of the
               node accessed through to_fillp */
            for  (scanf("%d", &data);
                  data != SENT;
                  scanf("%d", &data)) {
               newp = (list_node_t *)malloc(sizeof (list_node_t));
               newp->digit = data;
               to_fillp->restp = newp;
               to_fillp = newp;
            }

            /* Stores NULL in final node's restp component */
            to_fillp->restp = NULL;
      }
      return (ansp);
}
```


FIGURE 13.22 Function search

```
/*
 *  Searches a list for a specified target value. Returns a pointer to
 *  the first node containing target if found. Otherwise returns NULL.
 */
list_node_t *
search(list_node_t *headp,  /* input - pointer to head of list */
       int         target)  /* input - value to search for     */
{
     list_node_t *cur_nodep; /* pointer to node currently being checked */

     for  (cur_nodep = headp;
           cur_nodep != NULL  &&  cur_nodep->digit != target;
           cur_nodep = cur_nodep->restp) {}

     return (cur_nodep);
}
```

References

1. Problem Solving & Program Design in C, Jeri R. Hanly & Elliot B. Koffman, Pearson 8. Edition, Global Edition
2. C How to Program, Paul Deitel, Harvey Deitel. Pearson 8th Edition, Global Edition.
3. https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap
4. https://www.amazon.com/Pencil-Grip-Awareness-Development-DBD-965/dp/B003AZ7FE6