

Pointers and Dynamic Data Structures Chapter 13

Problem Solving & Program Design in C

Eighth Edition

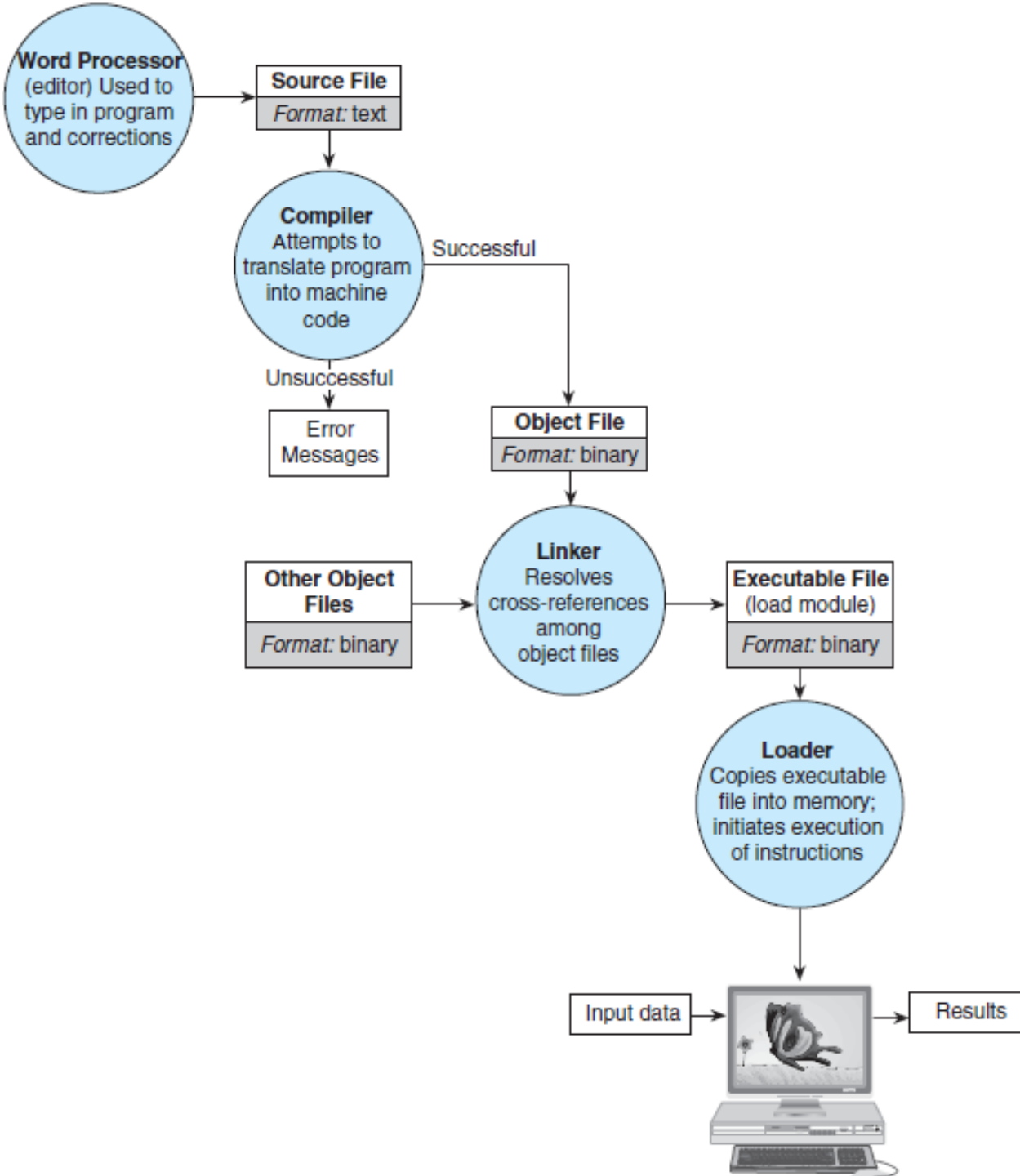
Jeri R. Hanly & Elliot B. Koffman

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

Previous uses of pointers...

- Reference to data
- Output parameters
- Arrays and strings
- File pointers



What happens when we run our executable file?



```
func1(int x) {  
    x += 1;  
    return(x);  
}
```

```
int main(void) {  
    int n = 10;  
    n = func1(n);  
    return(0);  
}
```

Stack memory



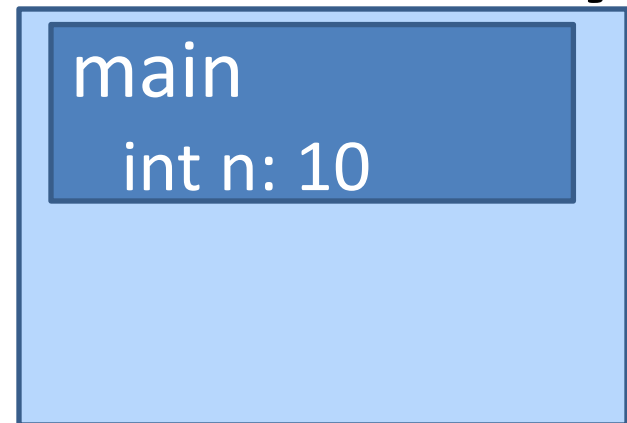
What happens when we run our executable file?



```
func1(int x) {  
    x += 1;  
    return(x);  
}
```

```
int main(void) {  
    int n = 10;  
    n = func1(n);  
    return(0);  
}
```

Stack memory



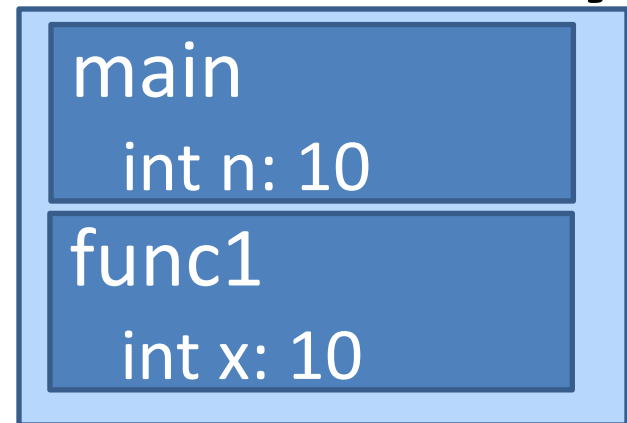
What happens when we run our executable file?



```
func1(int x) {  
    x += 1;  
    return(x);  
}
```

```
int main(void) {  
    int n = 10;  
    n = func1(n);  
    return(0);  
}
```

Stack memory



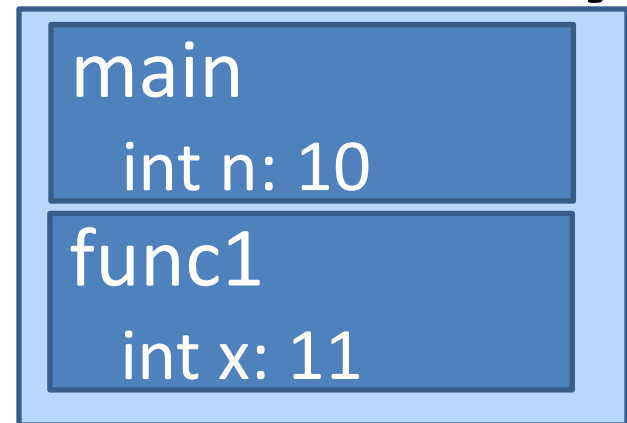
What happens when we run our executable file?



```
func1(int x) {  
    x += 1;  
    return(x);  
}
```

```
int main(void) {  
    int n = 10;  
    n = func1(n);  
    return(0);  
}
```

Stack memory



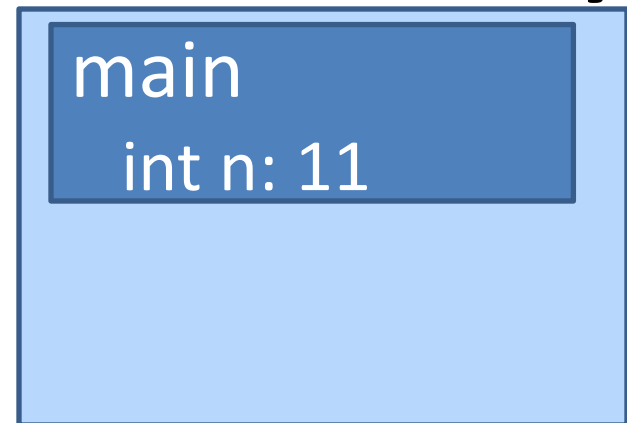
What happens when we run our executable file?



```
func1(int x) {  
    x += 1;  
    return(x);  
}
```

```
int main(void) {  
    int n = 10;  
    n = func1(n);  
    return(0);  
}
```

Stack memory



What happens when we run our executable file?



```
func1(int x) {  
    x += 1;  
    return(x);  
}
```

```
int main(void) {  
    int n = 10;  
    n = func1(n);  
    return(0);  
}
```

Stack memory



What happens when we run our executable file?



Stack memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    free(nump);  
}
```

Stack memory

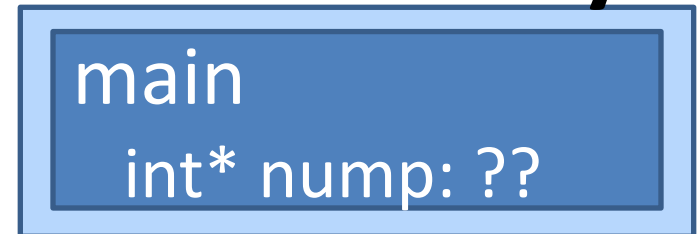
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    free(nump);  
}
```

Stack memory



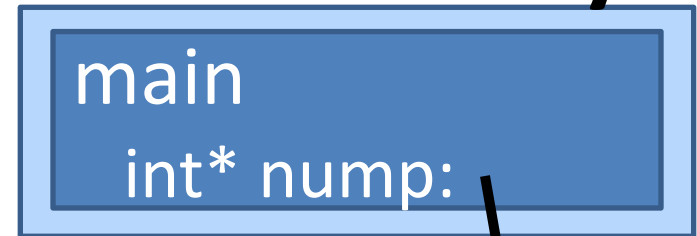
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    free(nump);  
}
```

Stack memory



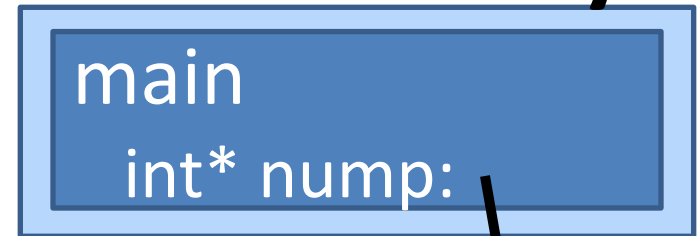
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    free(nump);  
}
```

Stack memory



int: 10

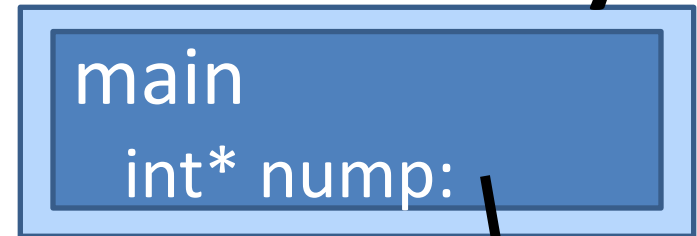
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    free(nump);  
}
```

Stack memory



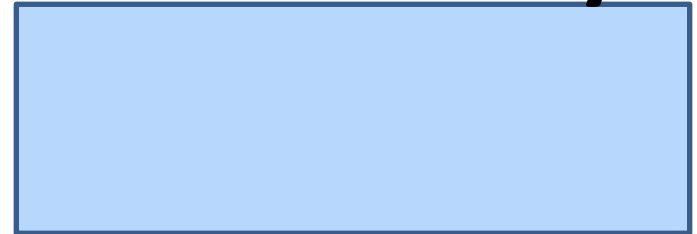
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    free(nump);  
}
```

Stack memory



Heap memory

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

Important functions

- malloc(<amnt of memory to reserve>)
- calloc(<num>, <amnt of memory to reserve>)
- free(pointer)

These are all from stdlib.h.

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    free(nump);  
}
```

Stack memory

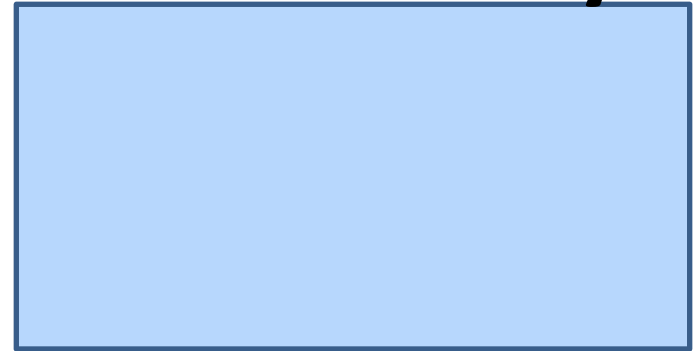
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    free(nump);  
}
```

Stack memory



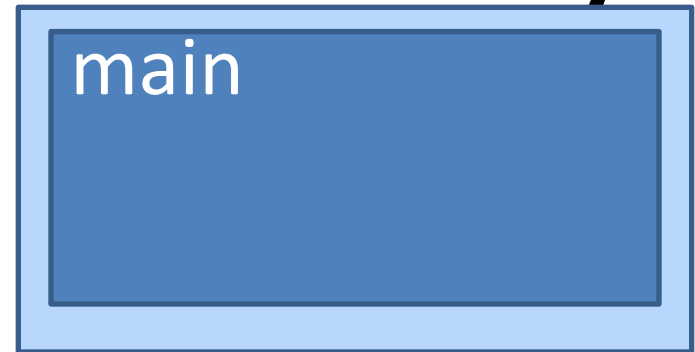
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    free(nump);  
}
```

Stack memory



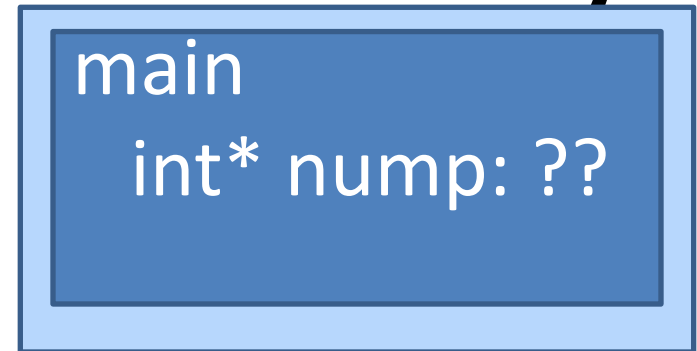
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    free(nump);  
}
```

Stack memory



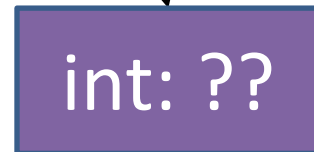
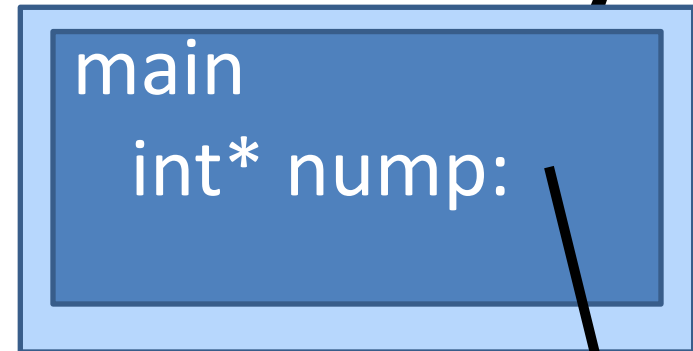
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    free(nump);  
}
```

Stack memory



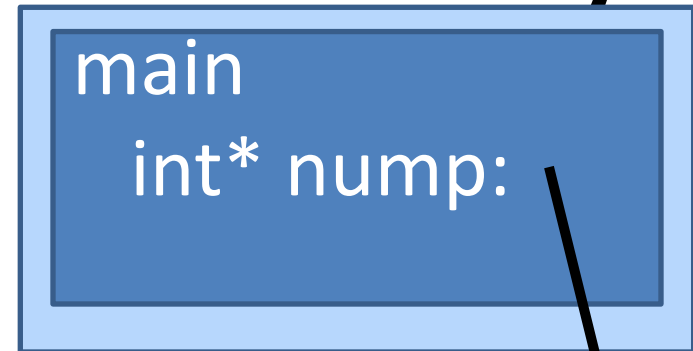
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    free(nump);  
}
```

Stack memory



int: 10

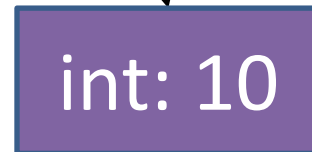
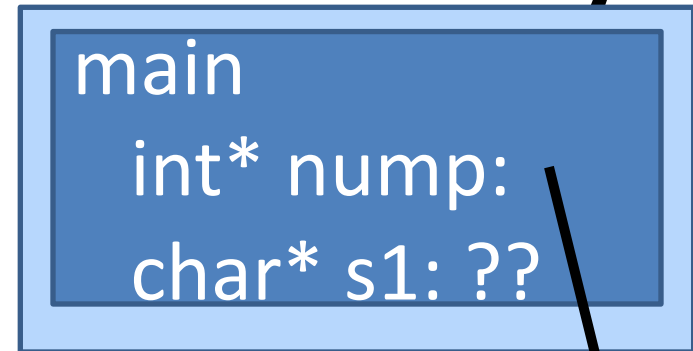
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    free(nump);  
}
```

Stack memory



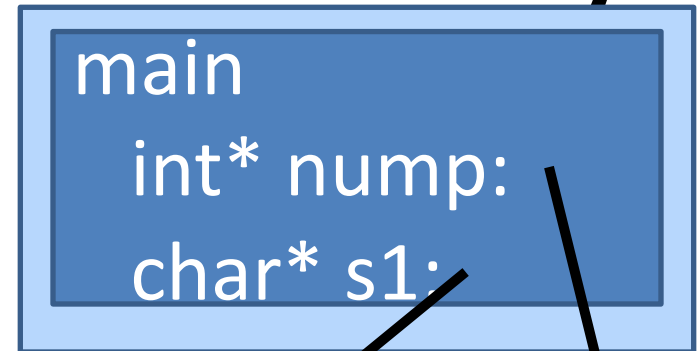
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    free(nump);  
}
```

Stack memory



int: 10



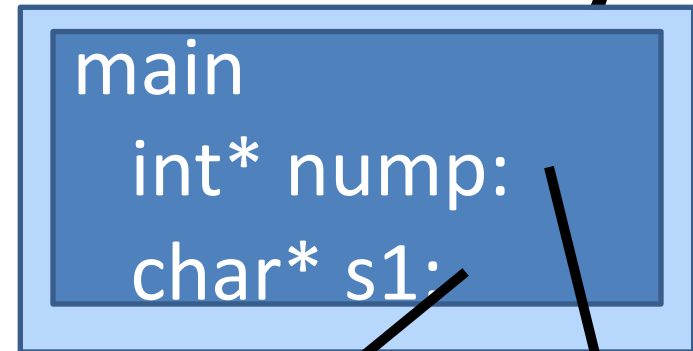
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    char string2[10] = "hi";  
    strcpy(string2, "aloha");  
    free(nump);  
}
```

Stack memory



int: 10

h e l l o / 0 ? ? ? ?

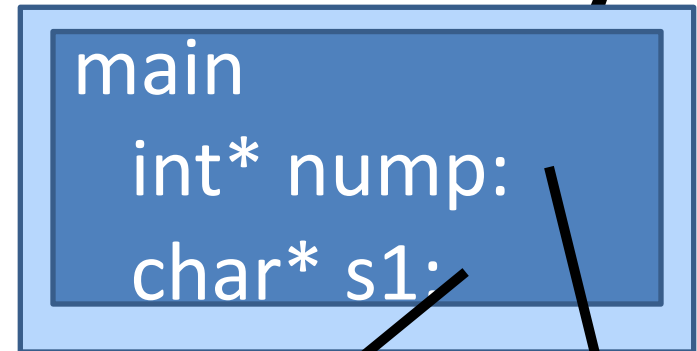
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    free(nump);  
}
```

Stack memory



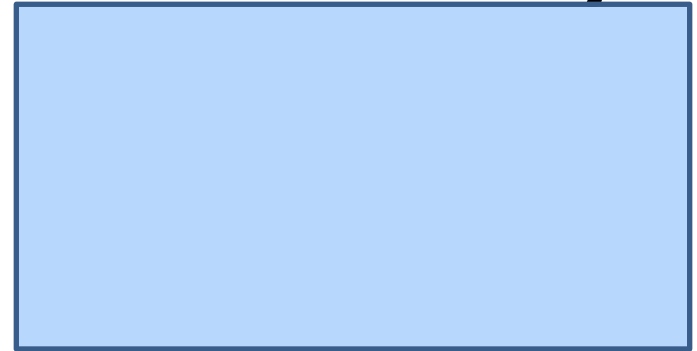
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    free(nump);  
}
```

Stack memory



h	e	l	l	o	/0	?	?	?	?
---	---	---	---	---	----	---	---	---	---

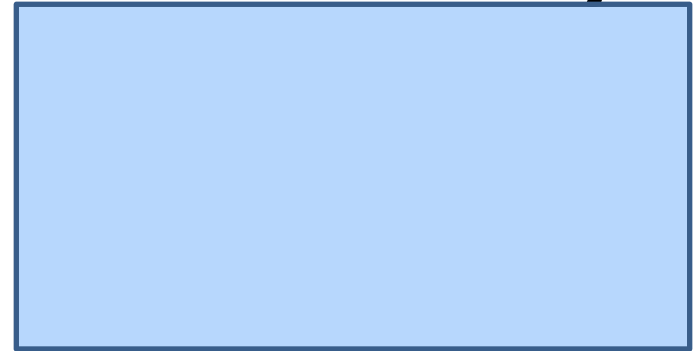
Heap memory

What happens when we run our executable file?



```
int main(void) {  
    int* nump;  
    nump = malloc(sizeof(int));  
    *nump = 10;  
    char* string1;  
    string1 = calloc(10, sizeof(char));  
    strcpy(string1, "hello");  
    free(nump);  
}
```

Stack memory



h	e	l	l	o	/0	?	?	?	?
---	---	---	---	---	----	---	---	---	---

Heap memory

Memory leaks

- When not all heap memory is freed before the end of a program
- Next time, we'll see a program (valgrind) that can check for memory leaks

(in reality, for a short-running program, not freeing our memory would be okay...but we want to be in the habit of freeing memory!)

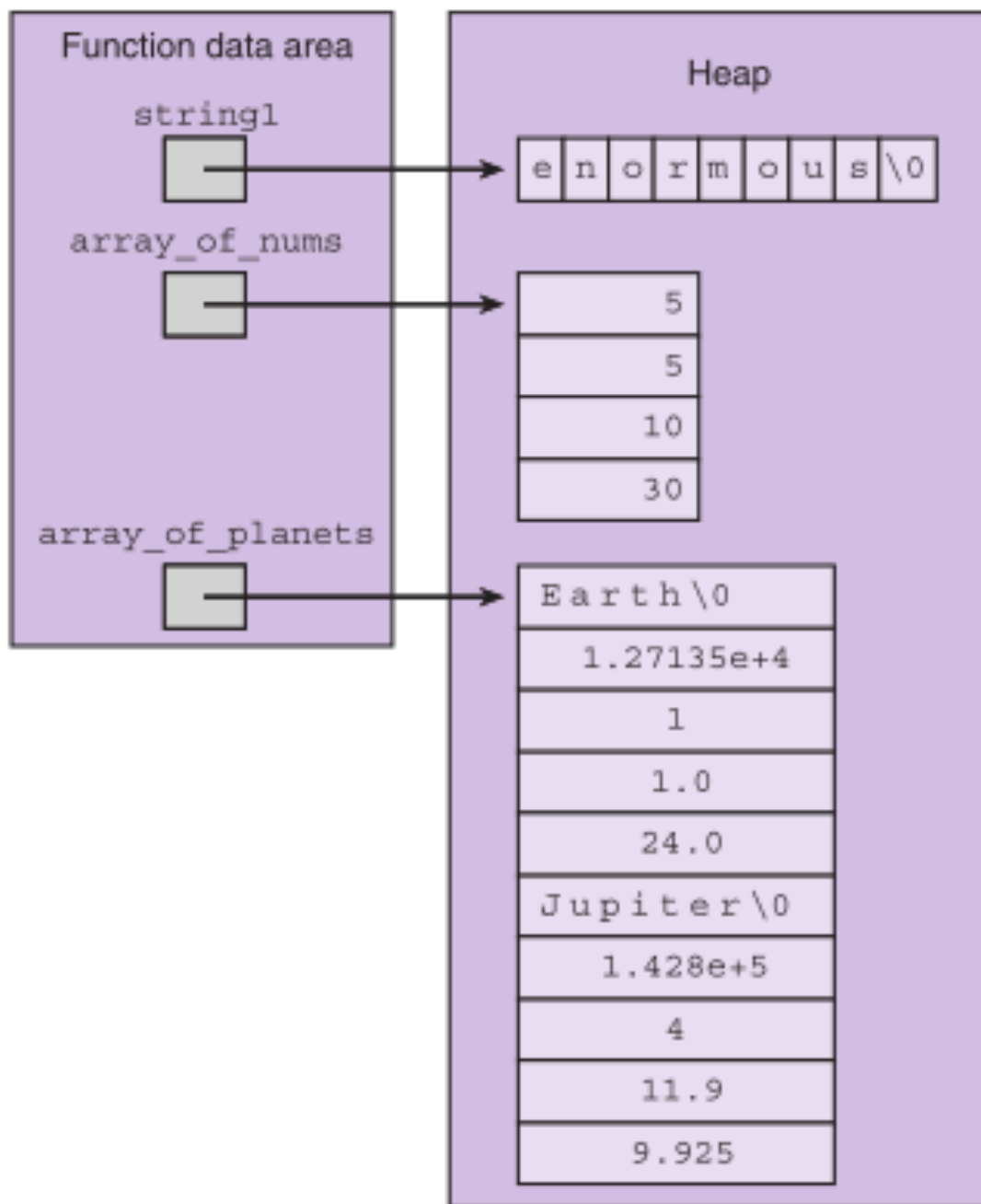
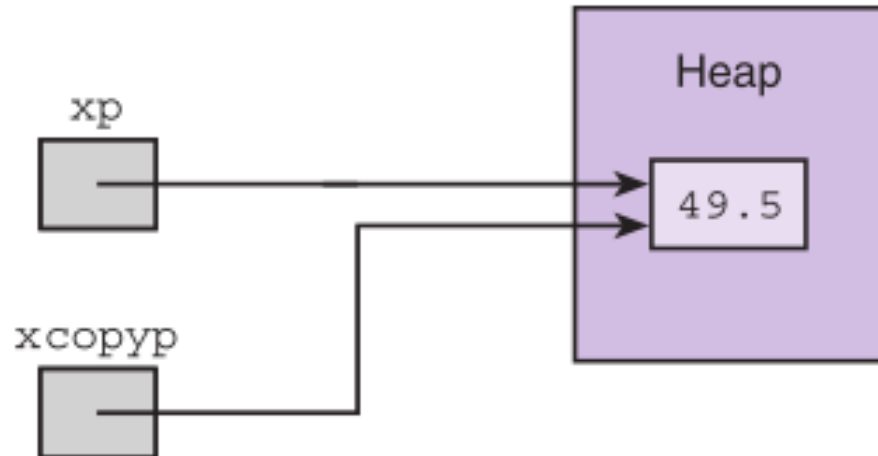


FIGURE 13.9

Multiple Pointers
to a Cell in the
Heap

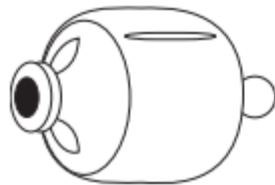


```
double *xp, *xcotyp;  
  
xp = (double *)malloc(sizeof (double));  
*xp = 49.5;  
xcotyp = xp;  
free(xp);  
. . .
```

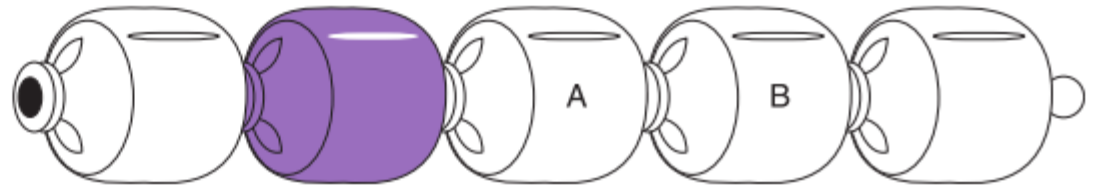
Linked Lists

- 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

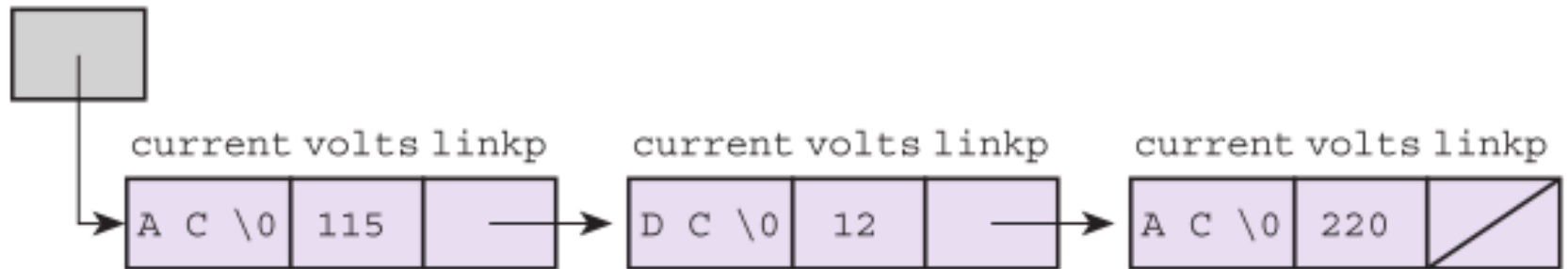
FIGURE 13.10 Children's Pop Beads in a Chain



Pop bead



Chain of pop beads



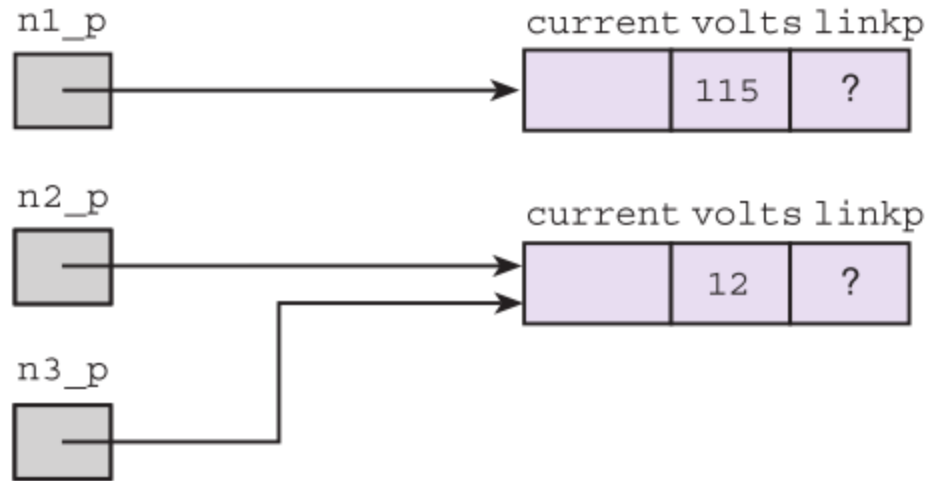


FIGURE 13.11

Multiple Pointers
to the Same
Structure

```
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;

n3_p = n2_p;
```

FIGURE 13.12

Linking Two Nodes

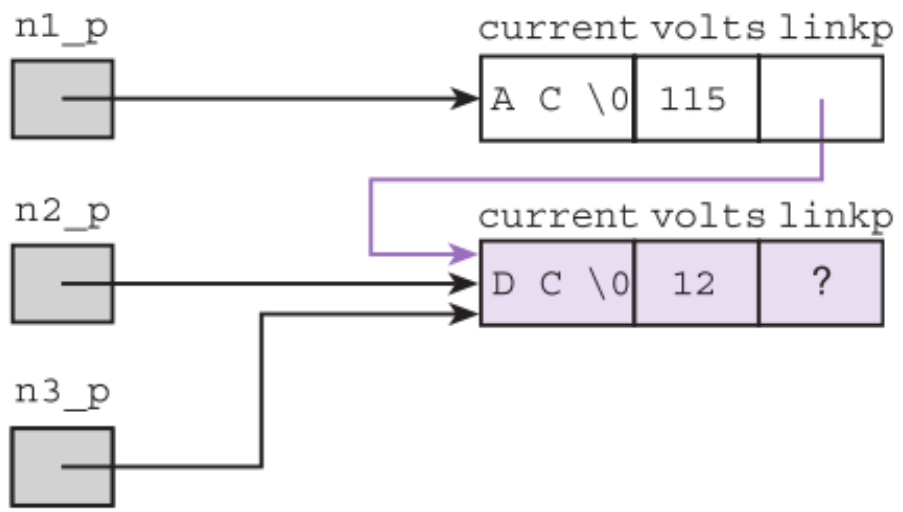
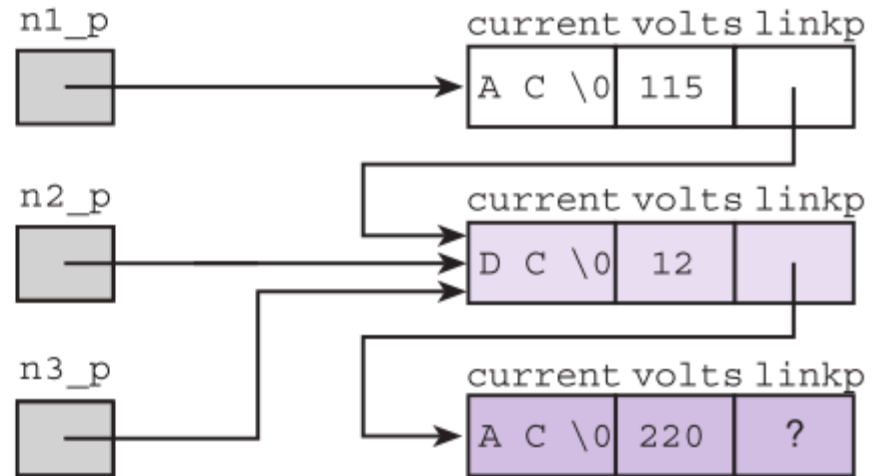


TABLE 13.2 Analyzing the Reference `n1_p->linkp->volts`

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

FIGURE 13.13

Three-Node Linked
List with Undefined
Final Pointer



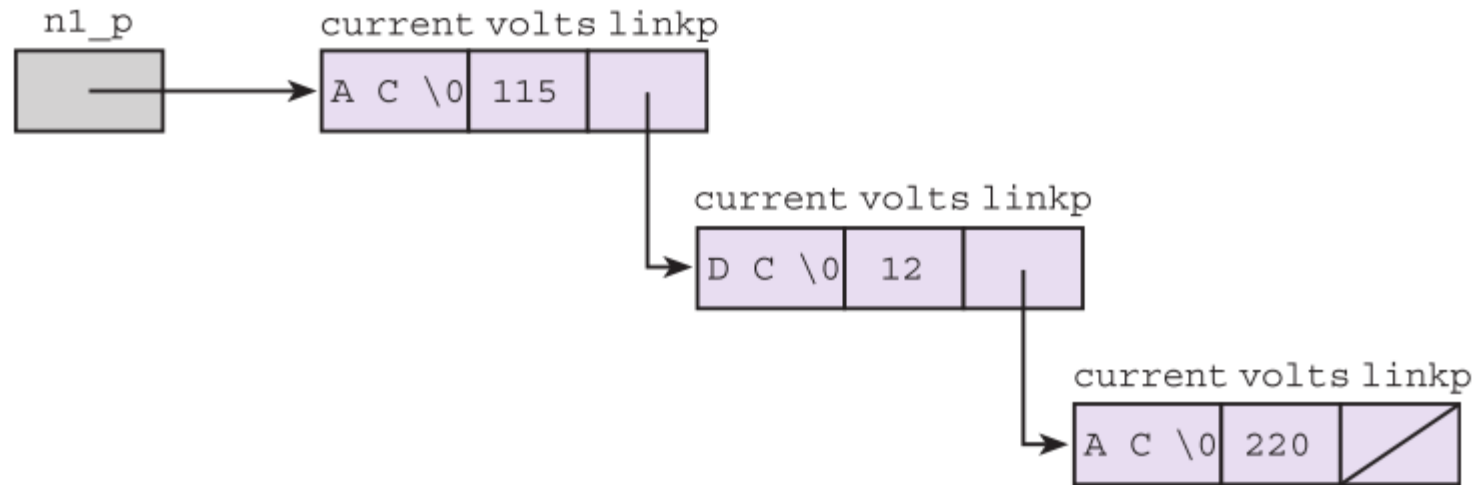


FIGURE 13.14

Three-Element
Linked List
Accessed Through
`n1_p`

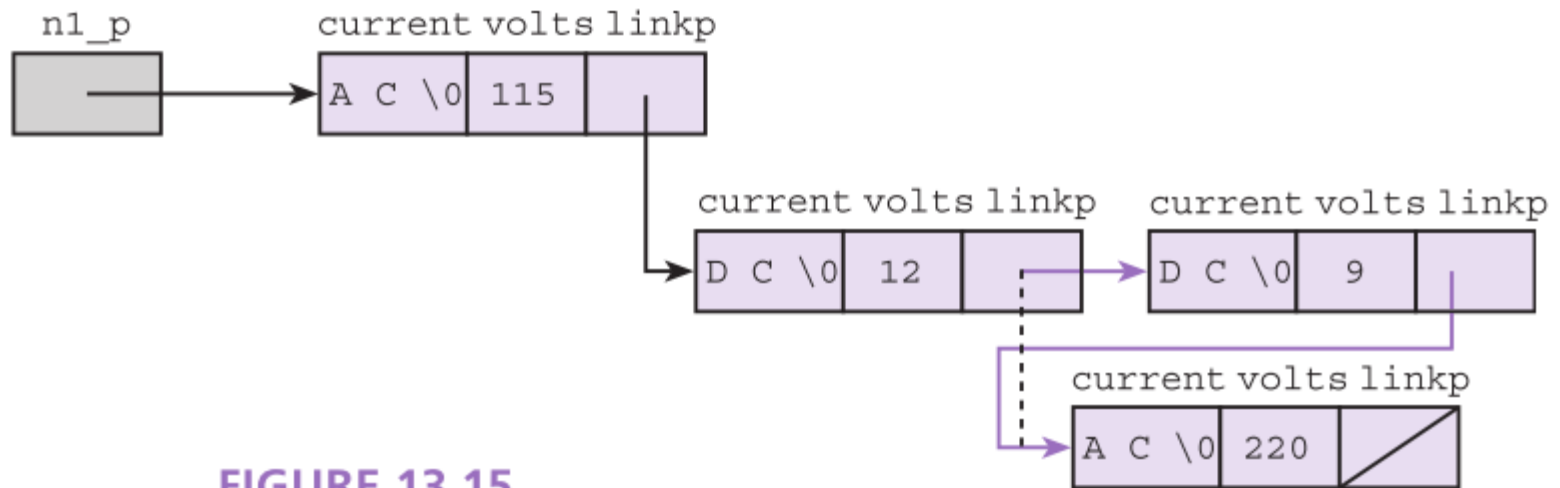
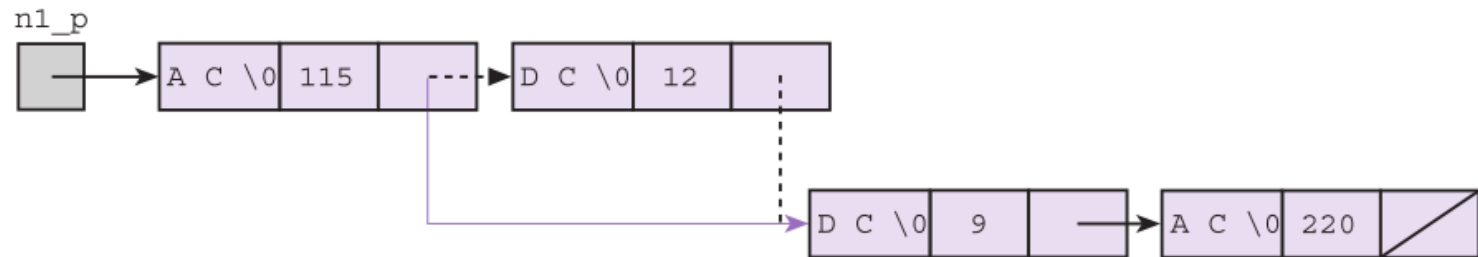


FIGURE 13.15

Linked List After
an Insertion

FIGURE 13.16 Linked List After a Deletion



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 13.17 Function `print_list`

```
1.  /*
2.   * Displays the list pointed to by headp
3.   */
4.  void
5.  print_list(list_node_t *headp)
6.  {
7.      if (headp == NULL) {      /* simple case - an empty list          */
8.          printf("\n");
9.      } else {                  /* recursive step - handles first element */
10.         printf("%d", headp->digit); /*          leaves rest to          */
11.         print_list(headp->restp); /*          recursion              */
12.     }
13. }
```

FIGURE 13.18 Comparison of Recursive and Iterative List Printing

<pre>/* 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); } }</pre>	<pre>{ 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"); }</pre>
---	--

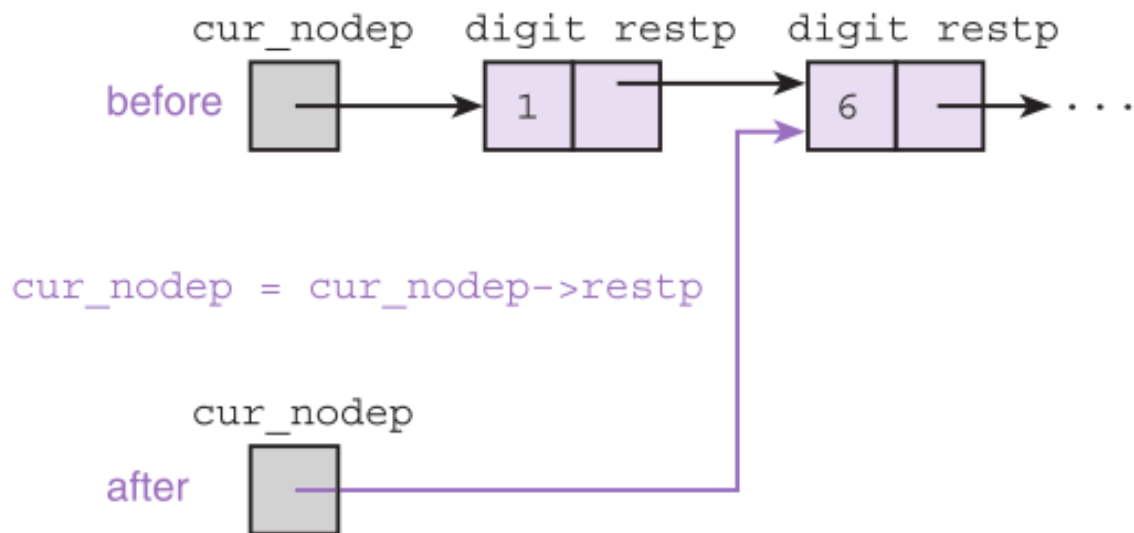


FIGURE 13.19

Update of
List-Traversing
Loop Control
Variable

FIGURE 13.20 Recursive Function `get_list`

```
1. #include <stdlib.h> /* gives access to malloc */
2. #define SENT -1
3. /*
4.  * Forms a linked list of an input list of integers
5.  * terminated by SENT
6.  */
7. list_node_t *
8. get_list(void)
9. {
10.     int data;
11.     list_node_t *ansp;
12.
13.     scanf("%d", &data);
14.     if (data == SENT) {
15.         ansp = NULL;
16.     } else {
17.         ansp = (list_node_t *)malloc(sizeof (list_node_t));
18.         ansp->digit = data;
19.         ansp->restp = get_list();
20.     }
21.
22.     return (ansp);
23. }
```


FIGURE 13.21 Iterative Function `get_list`

```
1.  /*
2.   *  Forms a linked list of an input list of integers terminated by SENT
3.   */
4.  list_node_t *
5.  get_list(void)
6.  {
7.      int data;
8.      list_node_t *ansp,
9.                  *to_fillp, /* pointer to last node in list whose
10.                          restp component is unfilled */
11.                  *newp;     /* pointer to newly allocated node */
12.
13.      /* Builds first node, if there is one */
14.      scanf("%d", &data);
15.      if (data == SENT) {
16.          ansp = NULL;
17.      } else {
18.          ansp = (list_node_t *)malloc(sizeof (list_node_t));
19.          ansp->digit = data;
20.          to_fillp = ansp;
21.
22.          /* Continues building list by creating a node on each
23.             iteration and storing its pointer in the restp component of the
24.             node accessed through to_fillp */
25.          for (scanf("%d", &data);
26.               data != SENT;
27.               scanf("%d", &data)) {
28.              newp = (list_node_t *)malloc(sizeof (list_node_t));
29.              newp->digit = data;
30.              to_fillp->restp = newp;
31.              to_fillp = newp;
32.          }
33.
34.          /* Stores NULL in final node's restp component */
35.          to_fillp->restp = NULL;
36.      }
37.      return (ansp);
38. }
```

FIGURE 13.22 Function search

```
1.  /*
2.   *   Searches a list for a specified target value. Returns a pointer to
3.   *   the first node containing target if found. Otherwise returns NULL.
4.   */
5.  list_node_t *
6.  search(list_node_t *headp, /* input - pointer to head of list */
7.         int          target) /* input - value to search for      */
8.  {
9.      list_node_t *cur_nodep; /* pointer to node currently being checked */
10.
11.      for (cur_nodep = headp;
12.           cur_nodep != NULL && cur_nodep->digit != target;
13.           cur_nodep = cur_nodep->restp) {}
14.
15.      return (cur_nodep);
16. }
```

Wrap Up

- Function `malloc` from the `stdlib` library can be used to allocate single elements, or nodes, or a dynamic data structure.
- Function `calloc` from `stdlib` dynamically allocates an array.
- Function `free` from `stdlib` returns memory cells to the storage heap.
- Linked lists can implement stacks, queues, and ordered lists

Appendix A: More about pointers

Pointer arithmetic