Pointers and Dynamic Data Structures
Chapter 13

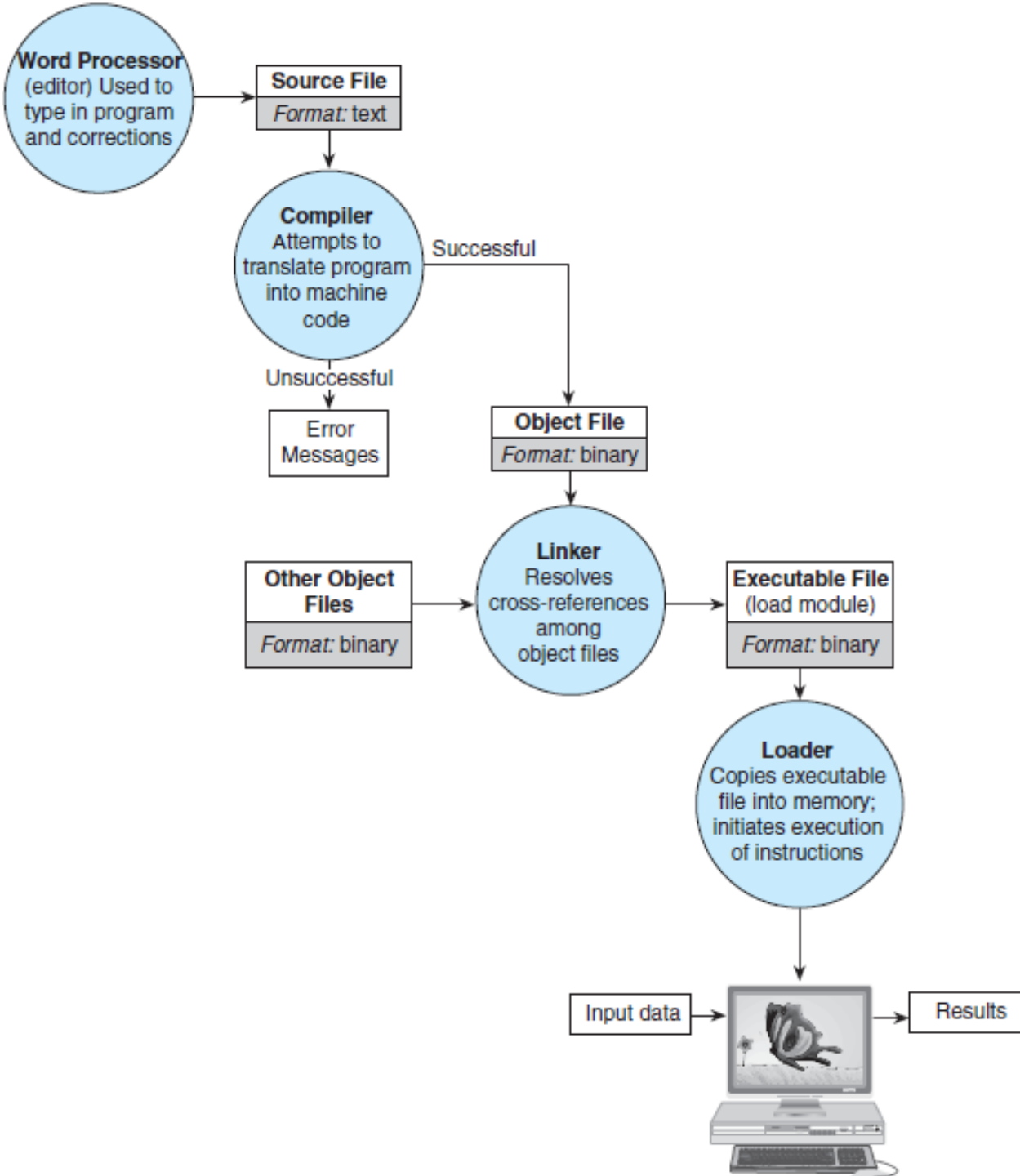*Problem Solving & Program Design in C*

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

```
 ┌─────────────────┐         ┌──────────────────┐
 │ Word Processor  │         │   Source File    │
 │ (editor) Used to│────────▶├──────────────────┤
 │ type in program │         │ Format: text     │
 │ and corrections │         └──────────────────┘
 └─────────────────┘                  │
                                      ▼
                          ┌──────────────────────┐
                          │      Compiler        │        Successful
                          │   Attempts to        ├──────────────────────┐
                          │ translate program    │                      │
                          │   into machine       │                      │
                          │      code            │                      │
                          └──────────────────────┘                      │
                                      │                                  ▼
                               Unsuccessful              ┌──────────────────┐
                                      ▼                  │   Object File    │
                          ┌──────────────┐               ├──────────────────┤
                          │    Error     │               │ Format: binary   │
                          │   Messages   │               └──────────────────┘
                          └──────────────┘                        │
                                                                  ▼
┌──────────────────┐          ┌────────────────────┐    ┌──────────────────┐
│  Other Object    │          │      Linker        │    │ Executable File  │
│     Files        │          │    Resolves        │    │  (load module)   │
├──────────────────┤─────────▶│ cross-references   ├───▶├──────────────────┤
│ Format: binary   │          │     among          │    │ Format: binary   │
└──────────────────┘          │   object files     │    └──────────────────┘
                              └────────────────────┘             │
                                                                 ▼
                                                      ┌────────────────────┐
                                                      │      Loader        │
                                                      │ Copies executable  │
                                                      │ file into memory;  │
                                                      │ initiates execution│
                                                      │ of instructions    │
                                                      └────────────────────┘
                                                                 │
                                                                 ▼
                  ┌────────────┐          ┌────────────┐       ┌──────────┐
                  │ Input data │─────────▶│  computer  │──────▶│ Results  │
                  └────────────┘          └────────────┘       └──────────┘
```

What happens when we run our executable file?

Input data → [monitor] → Results

```
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);
}
```
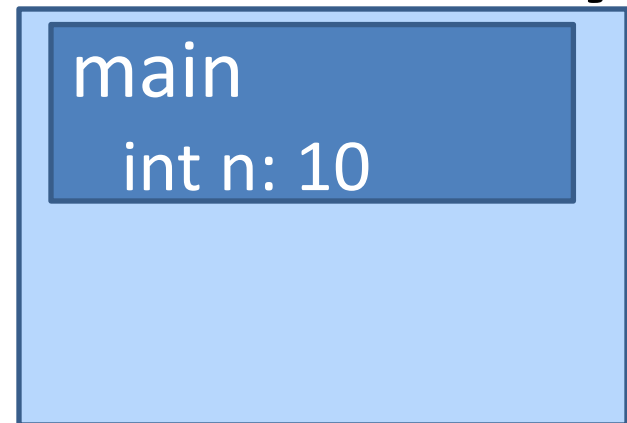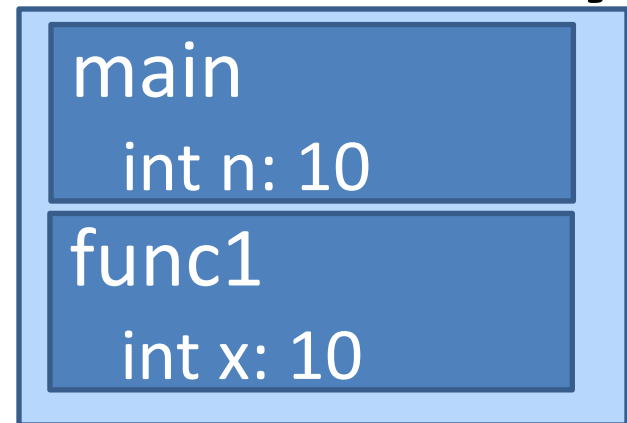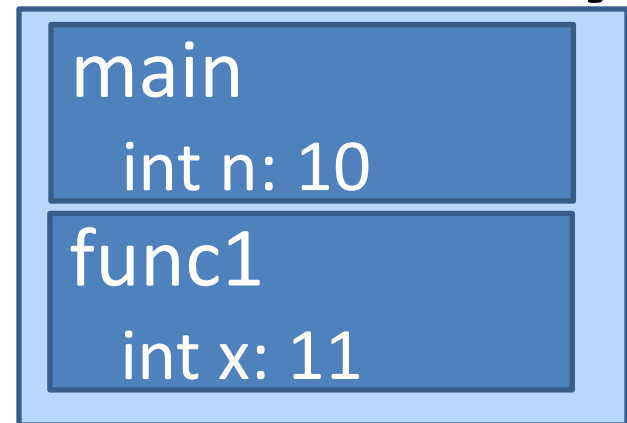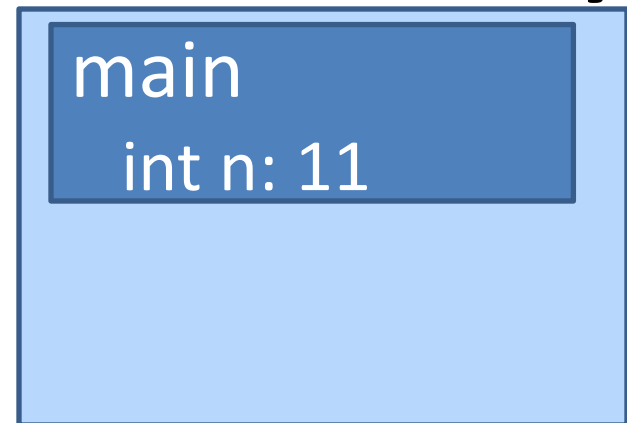
Stack memory

main
int n: 10

What happens when we run our executable file?

Input data → [computer] → Results

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

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

Stack memory

main
int n: 10

func1
int x: 10

What happens when we run our executable file?

Input data → Results

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

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

Stack memory

main
int n: 10
func1
int x: 11

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

main
int n: 11

What happens when we run our executable file?

Input data → [computer] → Results

```
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?

Input data → [computer] → Results

Stack memory

What happens when we run our executable file?

Input data → Results

Stack memory

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

Heap memory

What happens when we run our executable file?

Input data → Results

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

Stack memory

main
int* nump: ??

Heap memory

What happens when we run our executable file?

Stack memory

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

main
int* nump:

int: ??

Heap memory

What happens when we run our executable file?

Input data → [computer] → Results

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

Stack memory

main
int* nump:

int: 10

Heap memory

What happens when we run our executable file?

Input data → [screen] → Results

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

Stack memory

> main
> int* nump:

Heap memory

What happens when we run our executable file?

Input data → [computer] → Results

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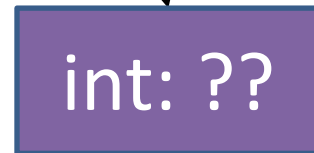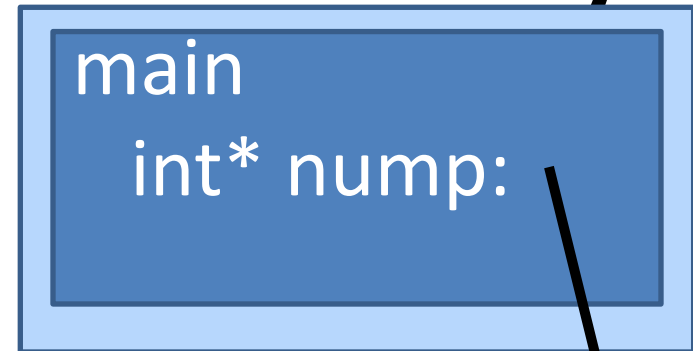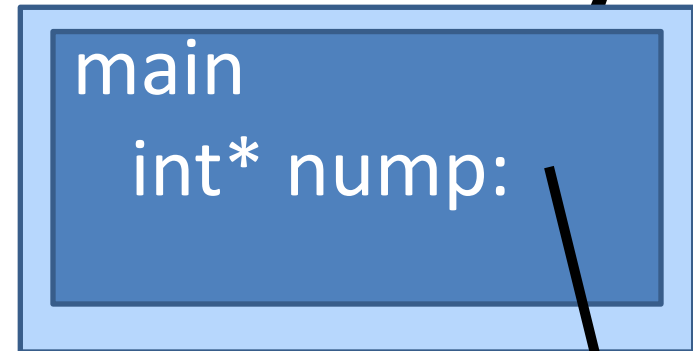
Stack memory

Heap memory

What happens when we run our executable file?

Input data → Results

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

Stack memory

main
int* nump:

Heap memory

What happens when we run our executable file?

Input data → [computer] → Results

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

Stack memory

main
int* nump:

undefined behavior!

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?

Input data → Results

Stack memory

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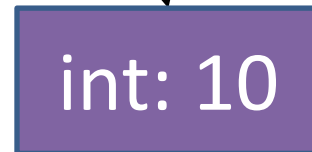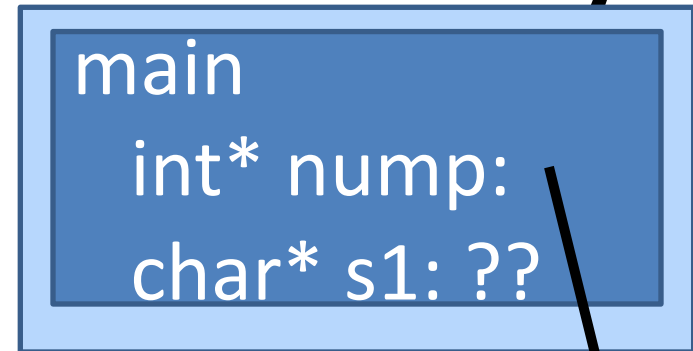
Heap memory

What happens when we run our executable file?

Input data → Results

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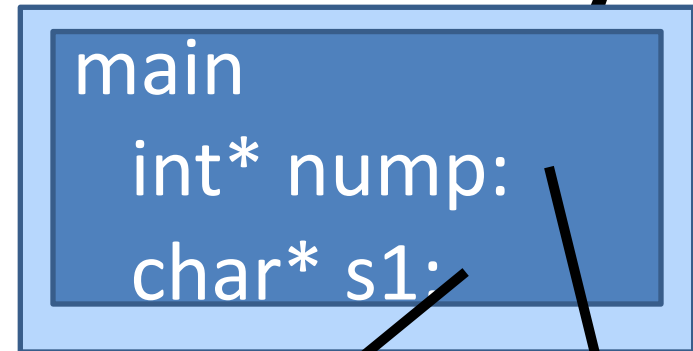
Stack memory

Heap memory

What happens when we run our executable file?

Input data → [computer] → Results

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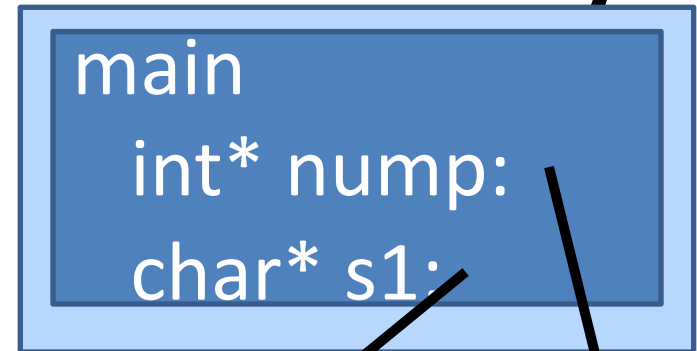
Stack memory

> main

Heap memory

What happens when we run our executable file?

Input data → [computer] → Results

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

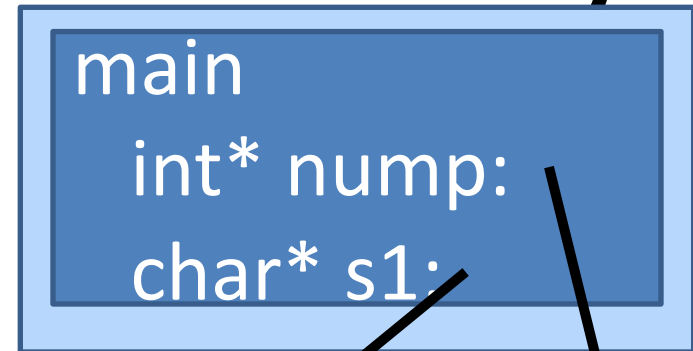## Stack memory

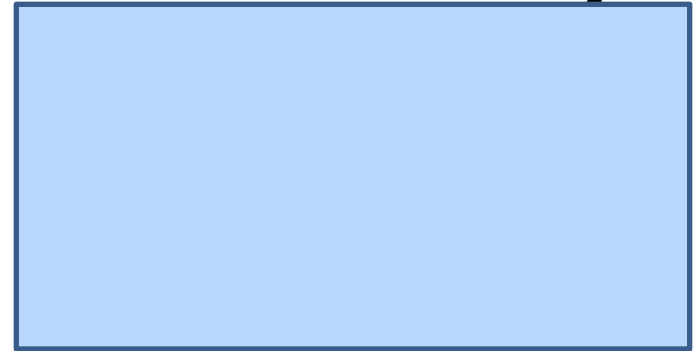main
   int* nump: ??

Heap memory

What happens when we run our executable file?

Stack memory

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

main
    int* nump:

int: ??

Heap memory

What happens when we run our executable file?

Input data → [computer] → Results

```c
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

> main
> int* nump:

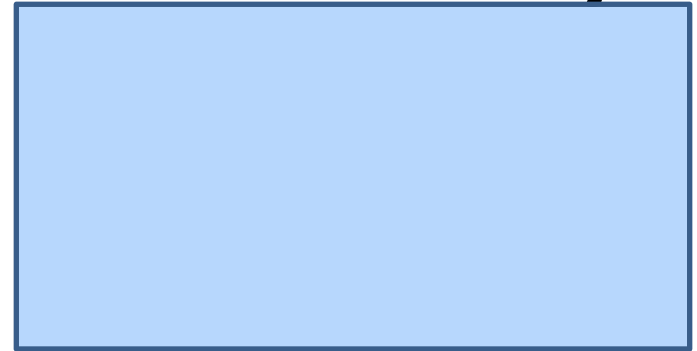int: 10

Heap memory

What happens when we run our executable file?

Input data → Results

```
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

main
int* nump:
char* s1: ??

int: 10

Heap memory

What happens when we run our executable file?

Input data → [computer] → Results

```c
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

main
int* nump:
char* s1:

int: 10

? ? ? ? ? ? ? ? ? ?

Heap memory

What happens when we run our executable file?

Input data → Results

```
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

main
  int* nump:
  char* s1:

int: 10

?	?	?	?	?	?	?	?	?	?

Heap memory

What happens when we run our executable file?

Input data → [computer] → Results

```
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

main
int* nump:
char* s1:

int: 10

| h | e | l | l | o | /0 | ? | ? | ? | ? |

Heap memory

What happens when we run our executable file?

Input data → Results

```
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?

Input data → Results

```
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

- If 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
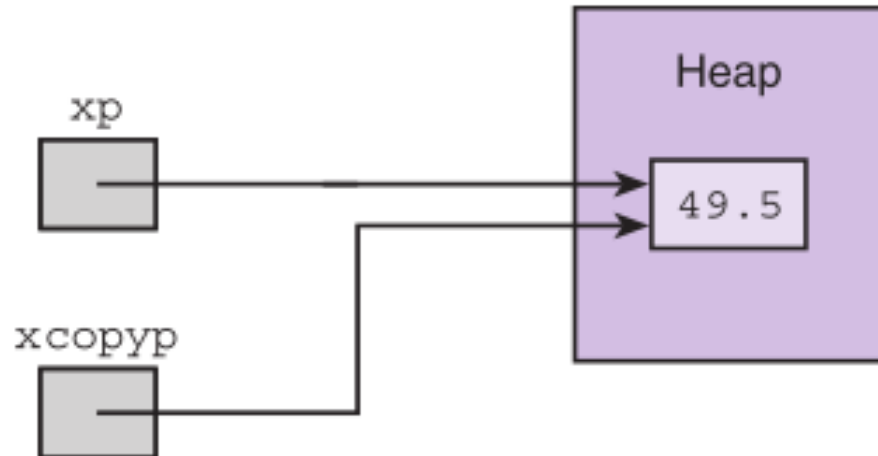

```
double *xp, *xcopyp;

xp = (double *)malloc(sizeof (double));
*xp = 49.5;
xcopyp = 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
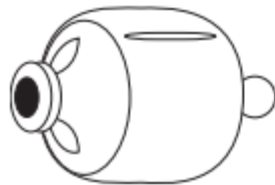
Pop bead

Chain of pop beads

current volts linkp

| A C \0 | 115 | → |

current volts linkp

| D C \0 | 12 | → |

current volts linkp

| A C \0 | 220 | |
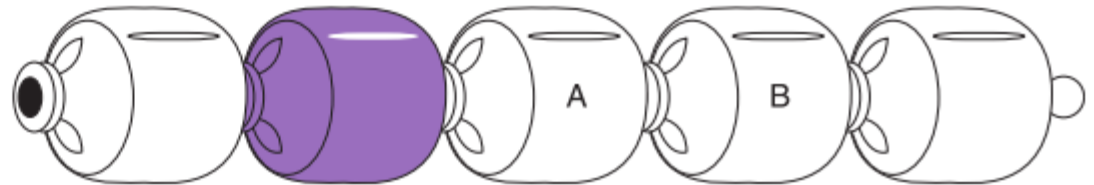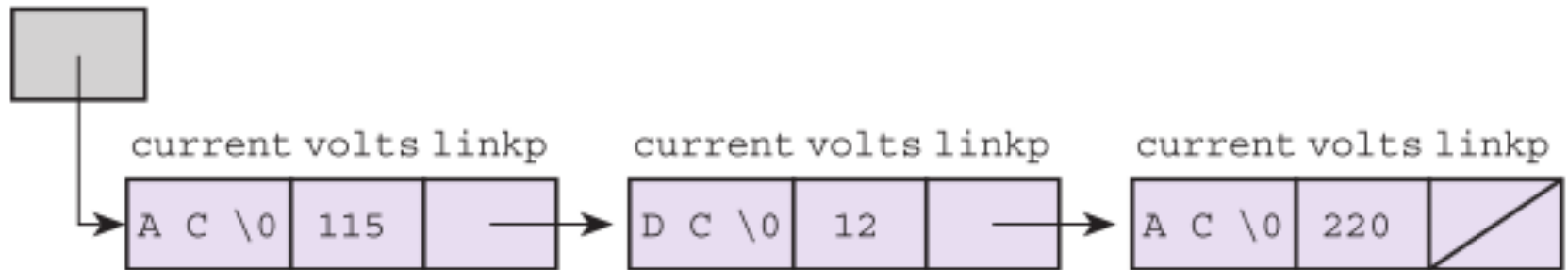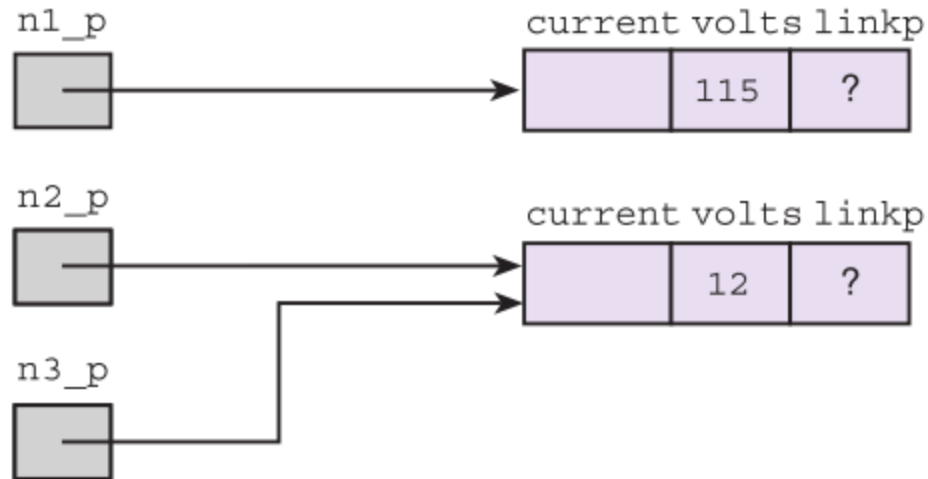
FIGURE 13.11

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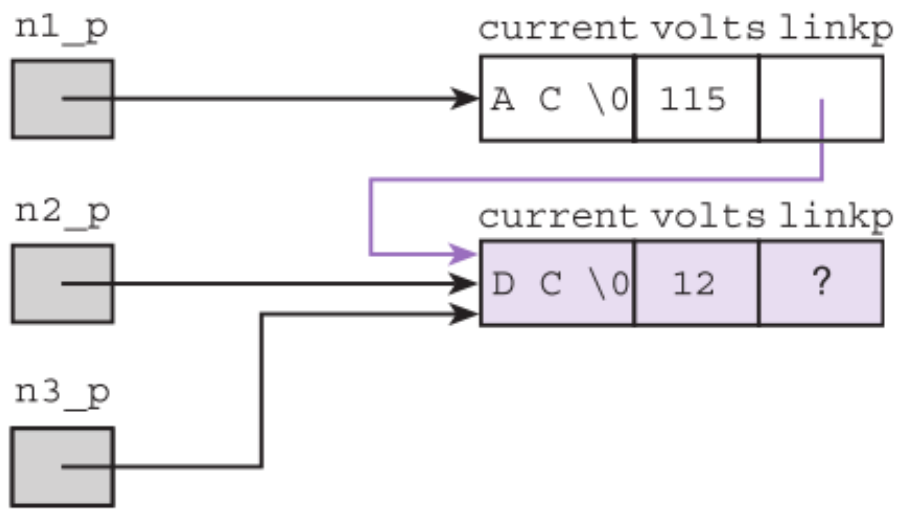
TABLE 13.2 Analyzing the Reference n1_p->linkp->volts

Section of Reference	Meaning
`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.

FIGURE 13.13

Three-Node Linked List with Undefined Final Pointer
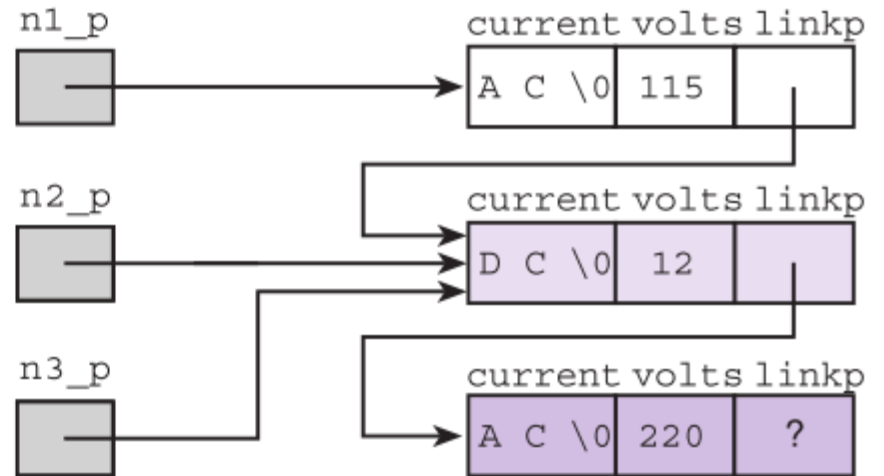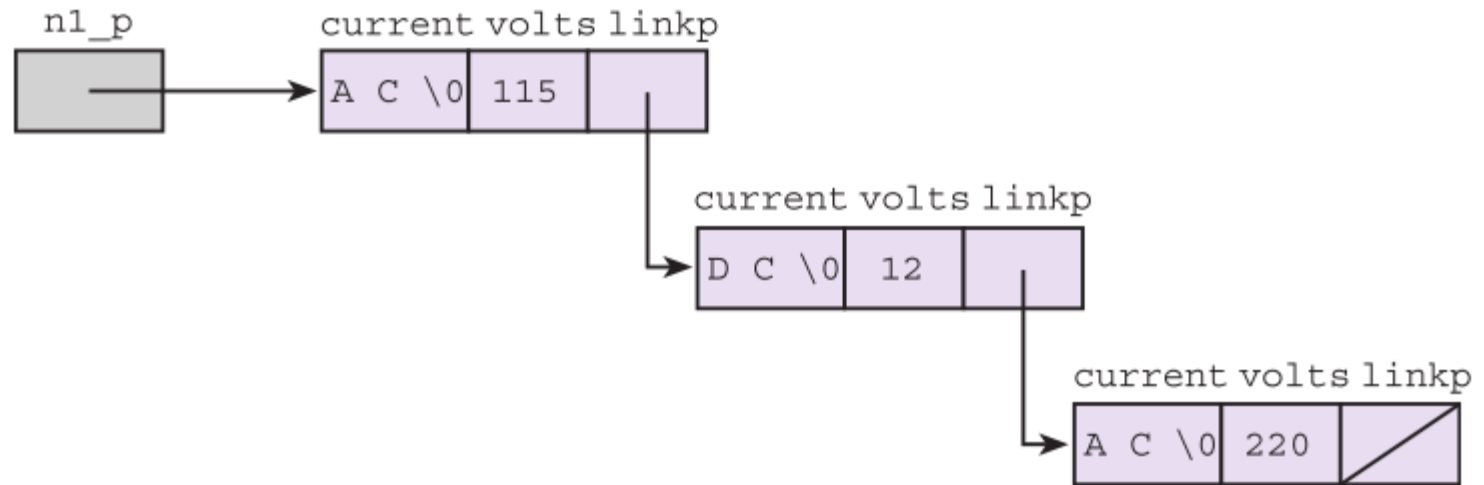
FIGURE 13.14

Three-Element
Linked List
Accessed Through
n1_p

```
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
        create_new_digit(2);
    head->next->next =
        create_new_digit(3);
}
```

Stack memory

Heap memory

43

```
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}


int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
            create_new_digit(2);
    head->next->next =
            create_new_digit(3);
}
```
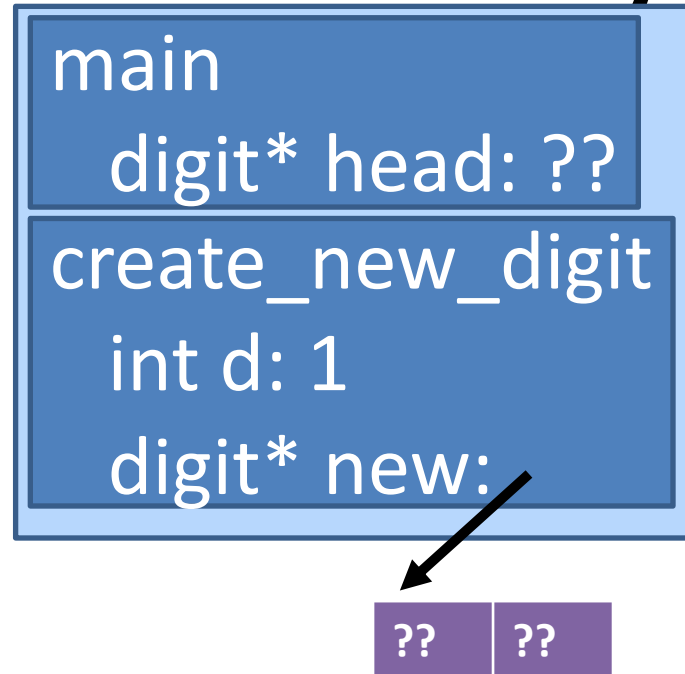
Stack memory

main

Heap memory

44

```c
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
            create_new_digit(2);
    head->next->next =
            create_new_digit(3);
}
```
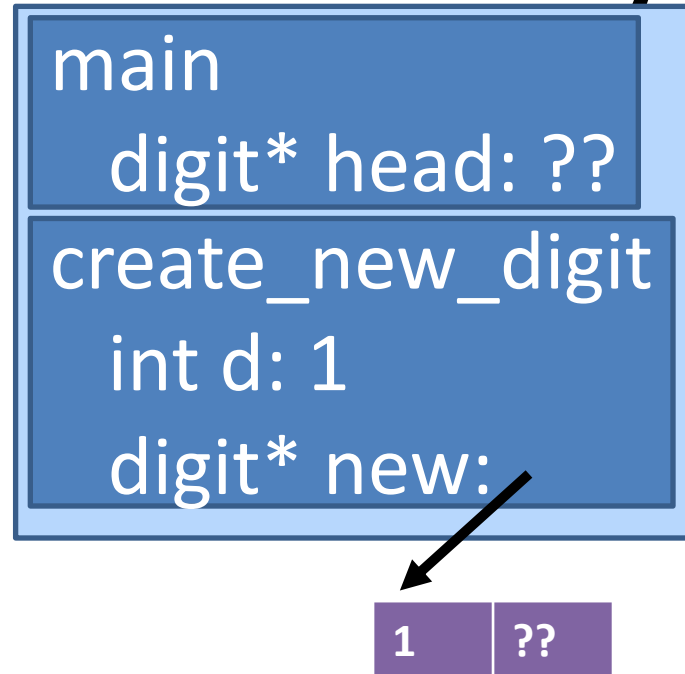
Stack memory

main
  digit* head: ??

Heap memory

```c
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
            create_new_digit(2);
    head->next->next =
            create_new_digit(3);
}
```

Stack memory

main
   digit* head: ??

create_new_digit
   int d: 1

Heap memory

```c
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
            create_new_digit(2);
    head->next->next =
            create_new_digit(3);
}
```
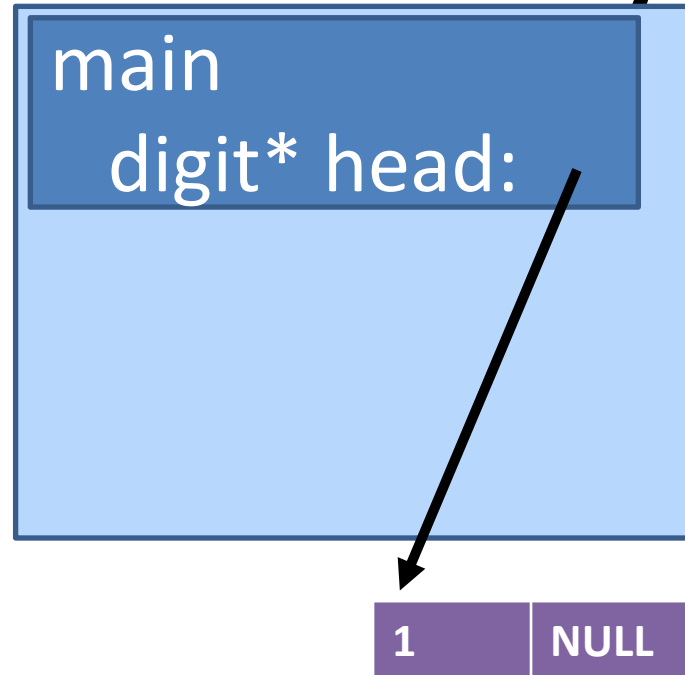
Stack memory

main
   digit* head: ??
create_new_digit
   int d: 1
   digit* new:

?? | ??

Heap memory

```c
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
            create_new_digit(2);
    head->next->next =
            create_new_digit(3);
}
```
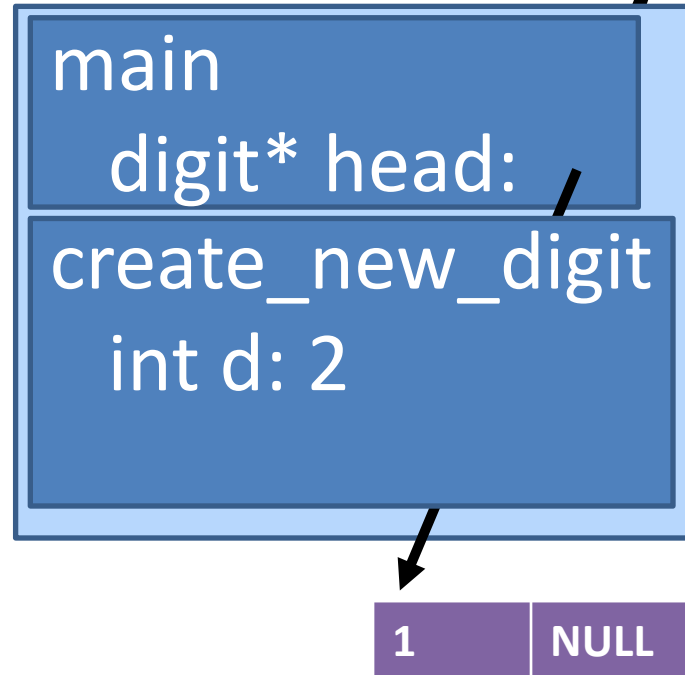
Stack memory

main
  digit* head: ??
create_new_digit
  int d: 1
  digit* new:

| 1 | ?? |

Heap memory

```
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
        create_new_digit(2);
    head->next->next =
        create_new_digit(3);
}
```
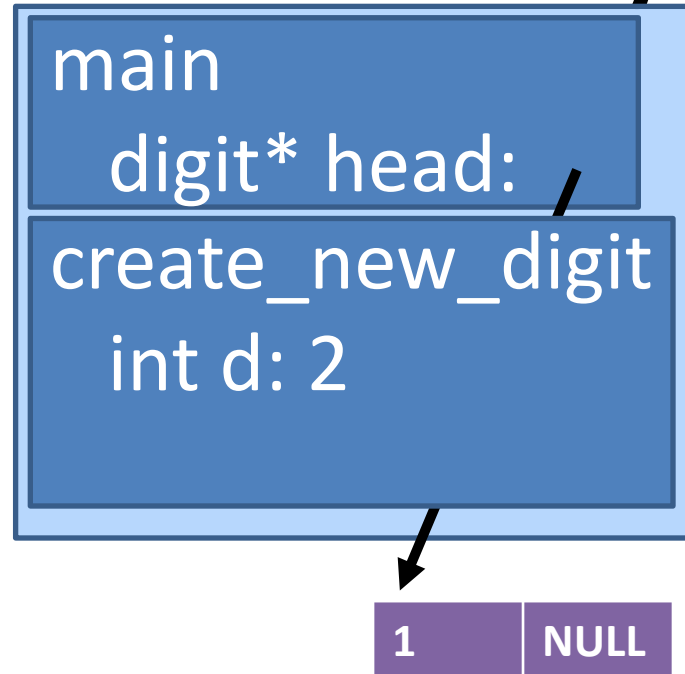
Stack memory

main
  digit* head: ??
create_new_digit
  int d: 1
  digit* new:

| 1 | NULL |

Heap memory

```c
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
            create_new_digit(2);
    head->next->next =
            create_new_digit(3);
}
```
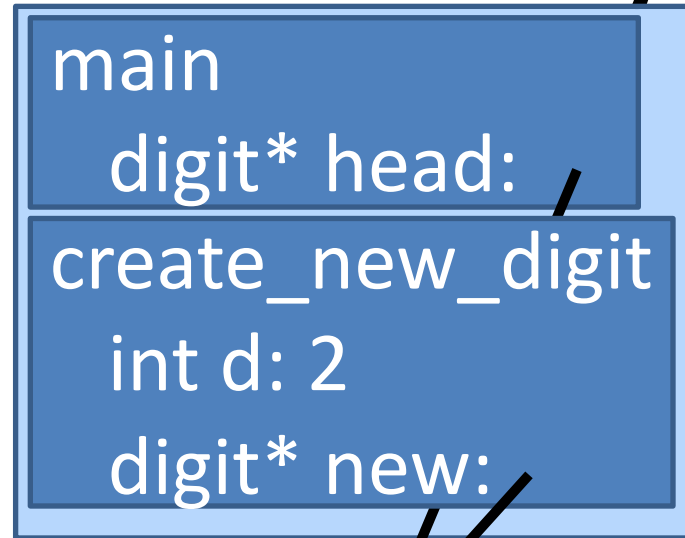
Stack memory

main
  digit* head:

| 1 | NULL |

Heap memory

```c
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
            create_new_digit(2);
    head->next->next =
            create_new_digit(3);
}
```
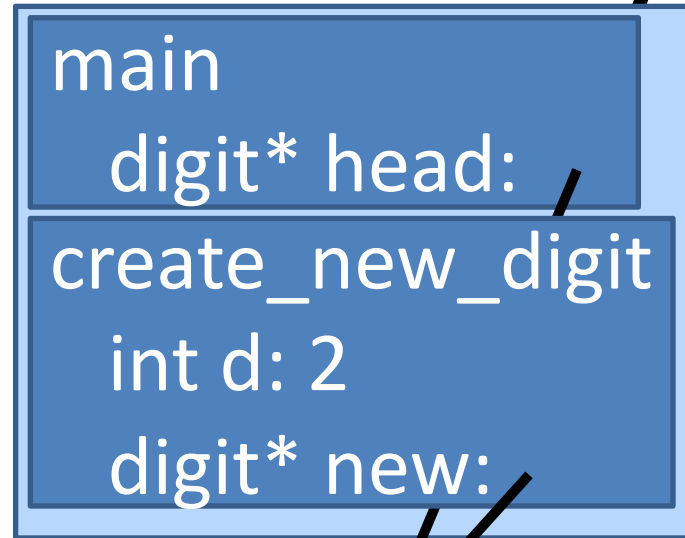
Stack memory

main
    digit* head:

create_new_digit
    int d: 2

| 1 | NULL |

Heap memory

```
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
            create_new_digit(2);
    head->next->next =
            create_new_digit(3);
}
```
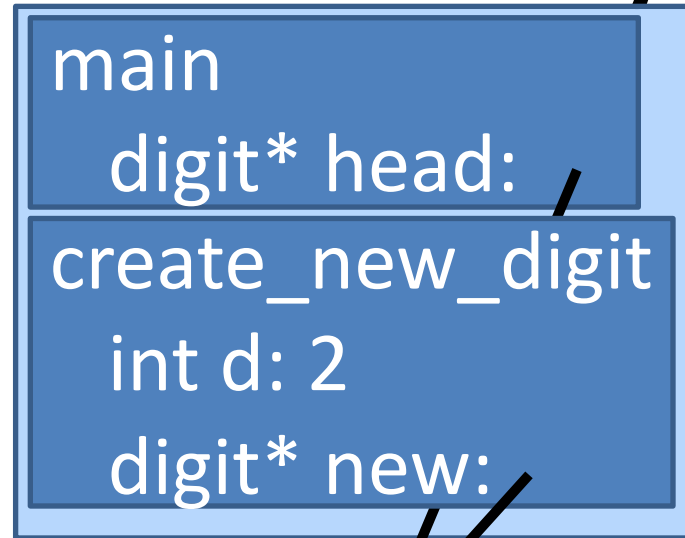
Stack memory

main
digit* head:

create_new_digit
int d: 2

1	NULL

Heap memory

```c
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
            create_new_digit(2);
    head->next->next =
            create_new_digit(3);
}
```
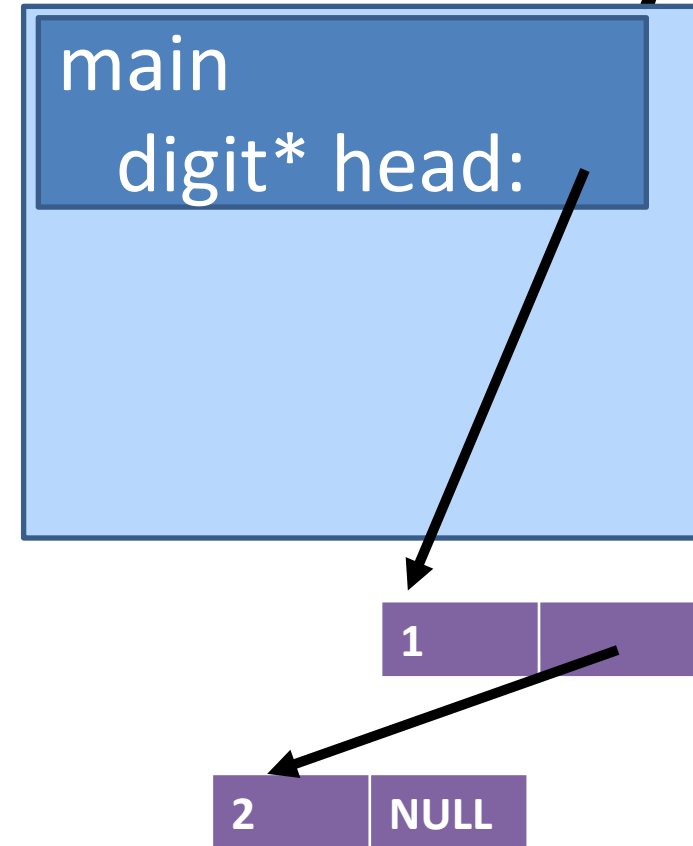
Stack memory

main
    digit* head:
create_new_digit
    int d: 2
    digit* new:

1	NULL

??	??

Heap memory

```
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
            create_new_digit(2);
    head->next->next =
            create_new_digit(3);
}
```
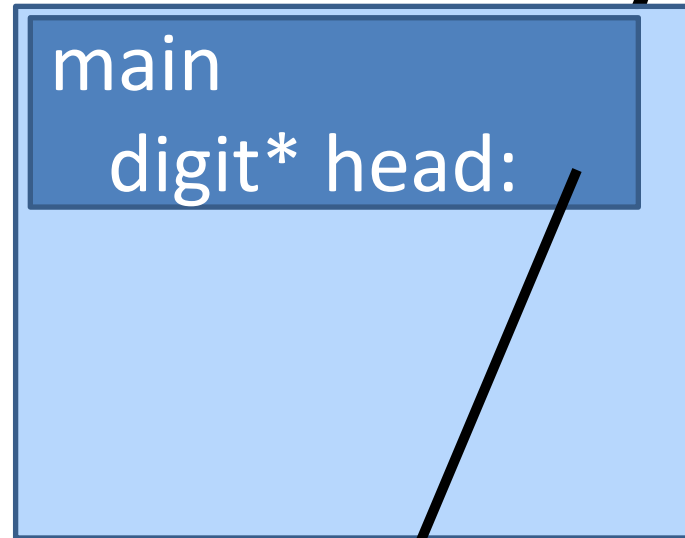
Stack memory

main
   digit* head:

create_new_digit
   int d: 2
   digit* new:

1	NULL

2	

Heap memory

```c
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
            create_new_digit(2);
    head->next->next =
            create_new_digit(3);
}
```
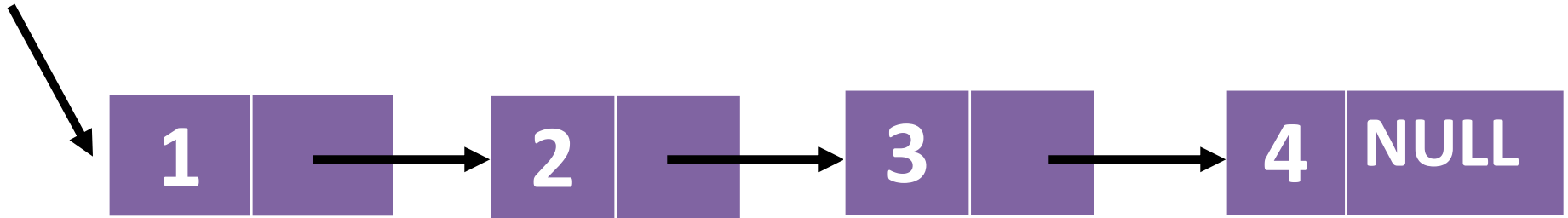
Stack memory

main
    digit* head:
create_new_digit
    int d: 2
    digit* new:

1	NULL

2	NULL

Heap memory

```c
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
        create_new_digit(2);
    head->next->next =
        create_new_digit(3);
}
```

Stack memory

main
    digit* head:

| 1 | |

| 2 | NULL |

Heap memory

```c
digit* create_new_digit(int d) {
    digit* new = malloc(sizeof(digit));
    new->d = d;
    new->next = NULL;
    return(new);
}

int main(void) {
    digit* head;
    head = create_new_digit(1);
    head->next =
        create_new_digit(2);
    head->next->next =
        create_new_digit(3);
}
```

Stack memory
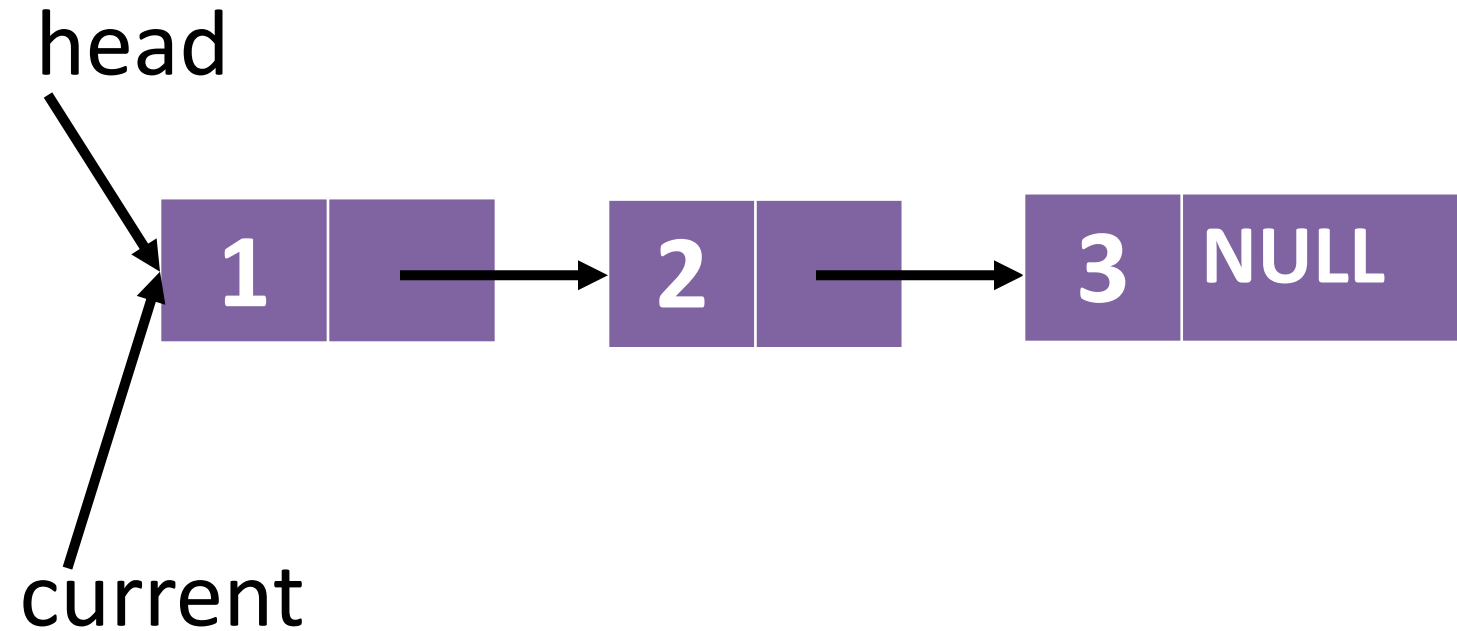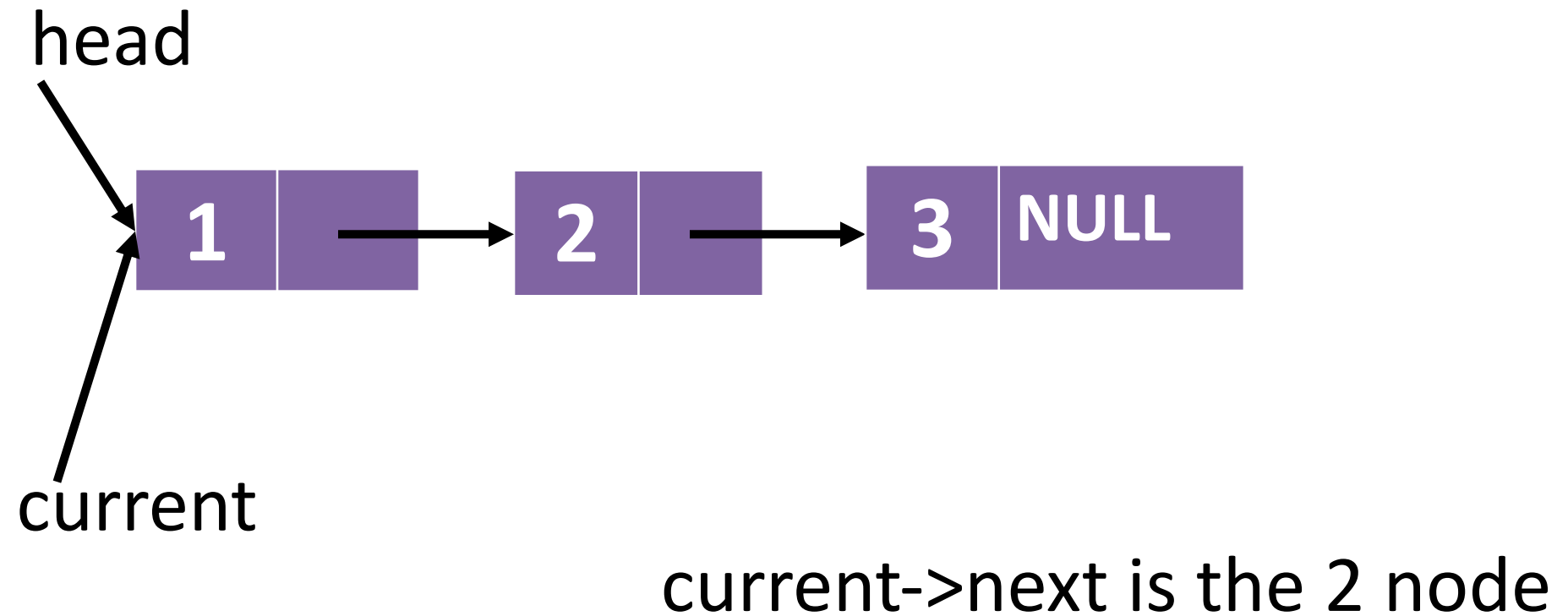
main
    digit* head:

1

2

3 | NULL

Heap memory

Inserting a node at end of list
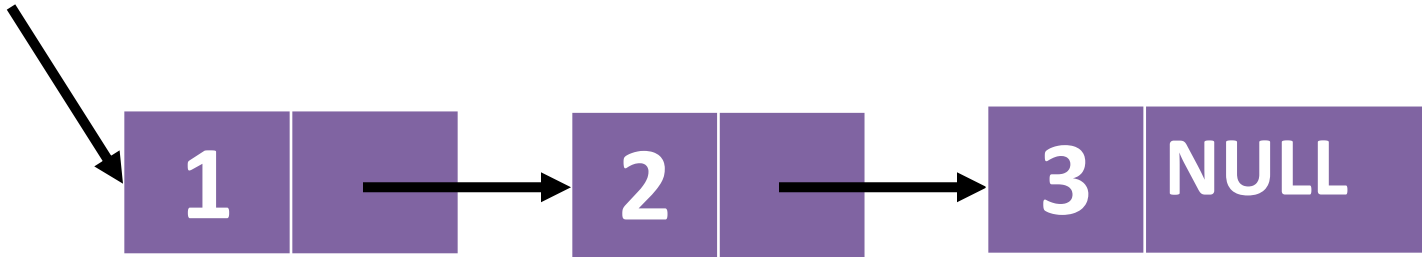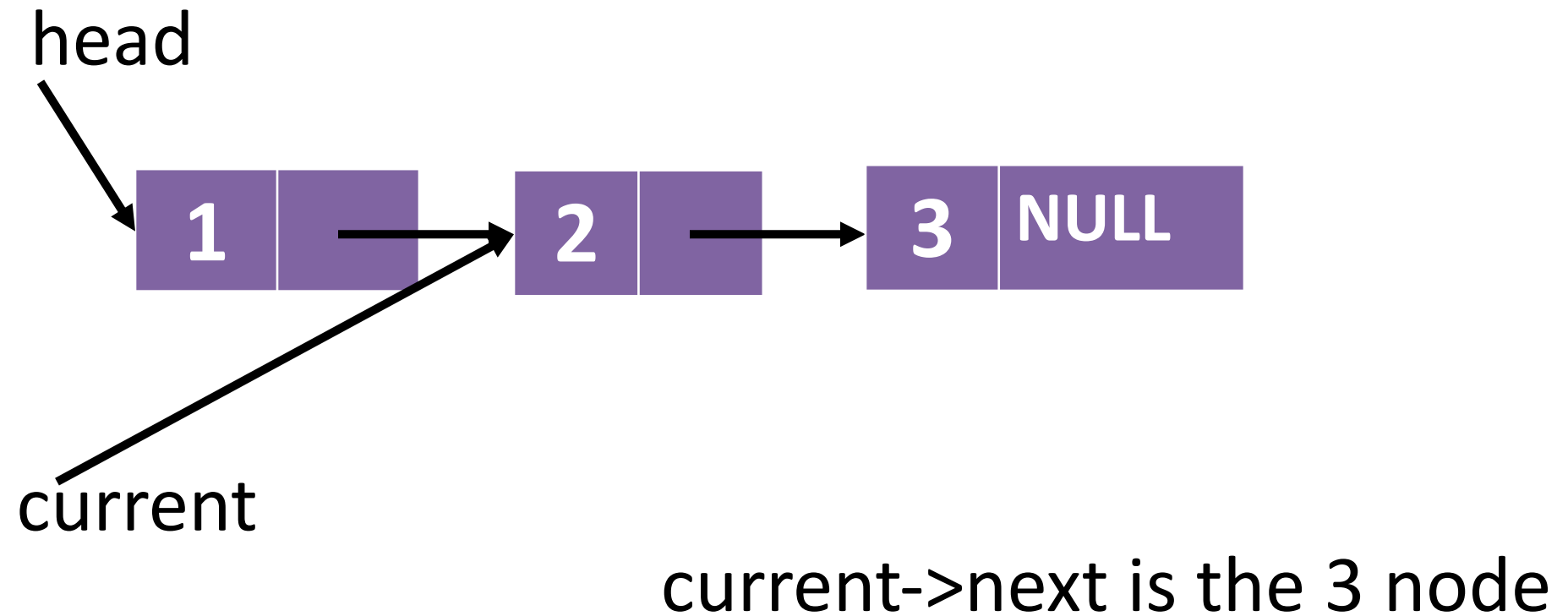
head

Inserting a node at end of list

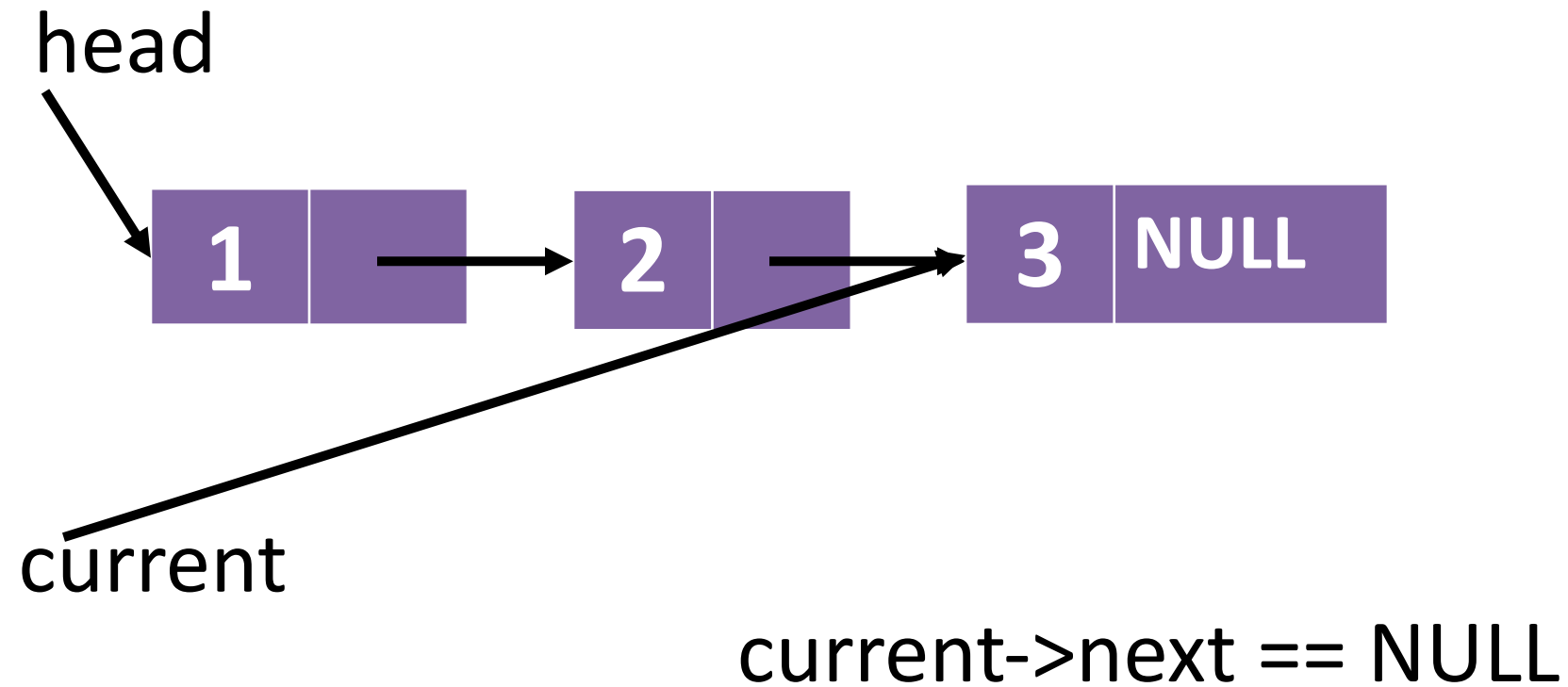head

Inserting a node at end of list

head

Inserting a node at end of list

head

1 → 2 → 3 NULL

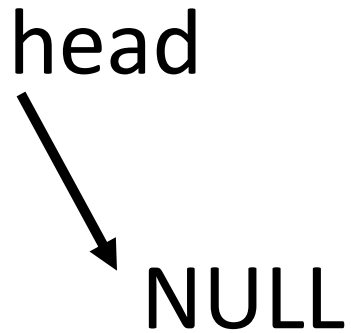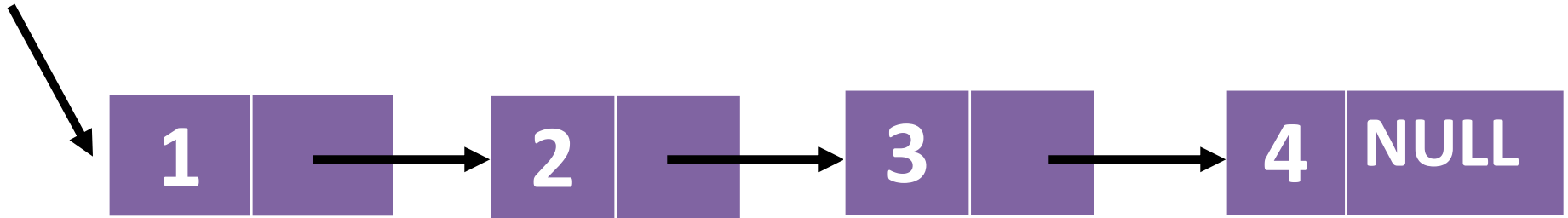current

Inserting a node at end of list

head

current

1 → 2 → 3 NULL

current->next is the 2 node

Inserting a node at end of list

head

current

1 → 2 → 3 NULL

current->next is the 3 node

Inserting a node at end of list

head

1 → 2 → 3 | NULL

current

current->next == NULL

Inserting a node at end of list

head

1 → 2 → 3 → 4 NULL
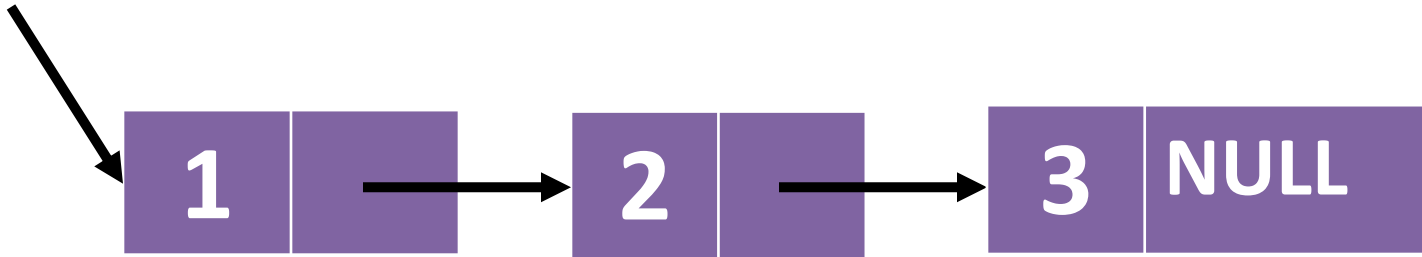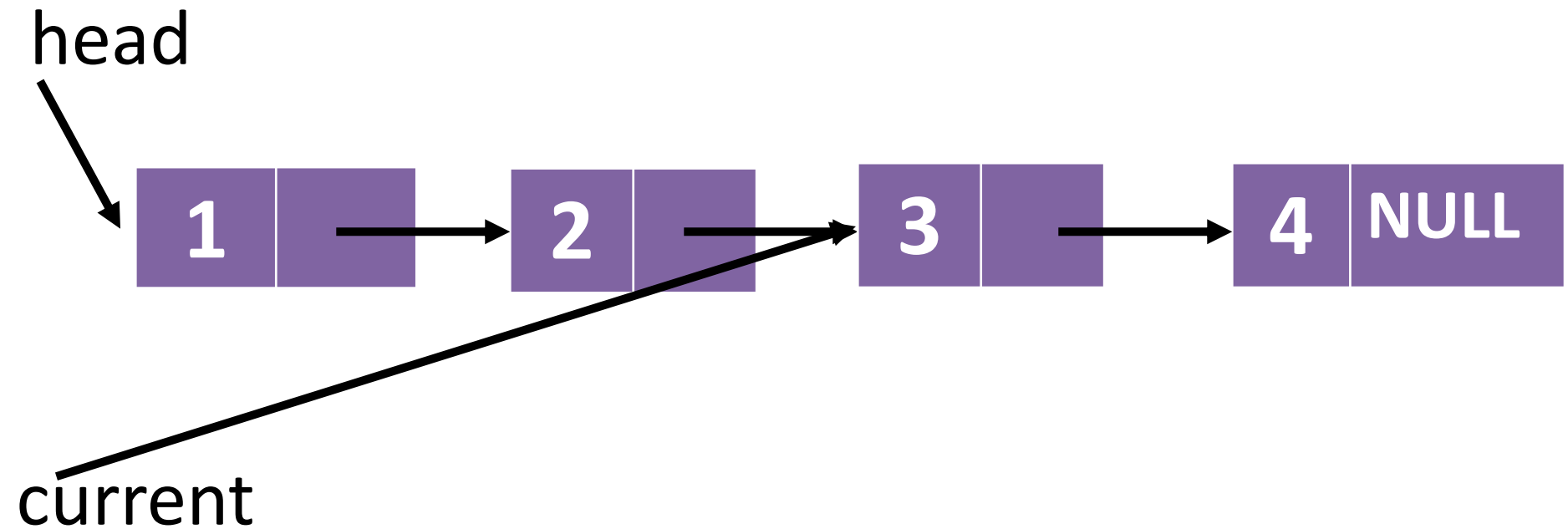
current

current->next = new

What if list is empty?
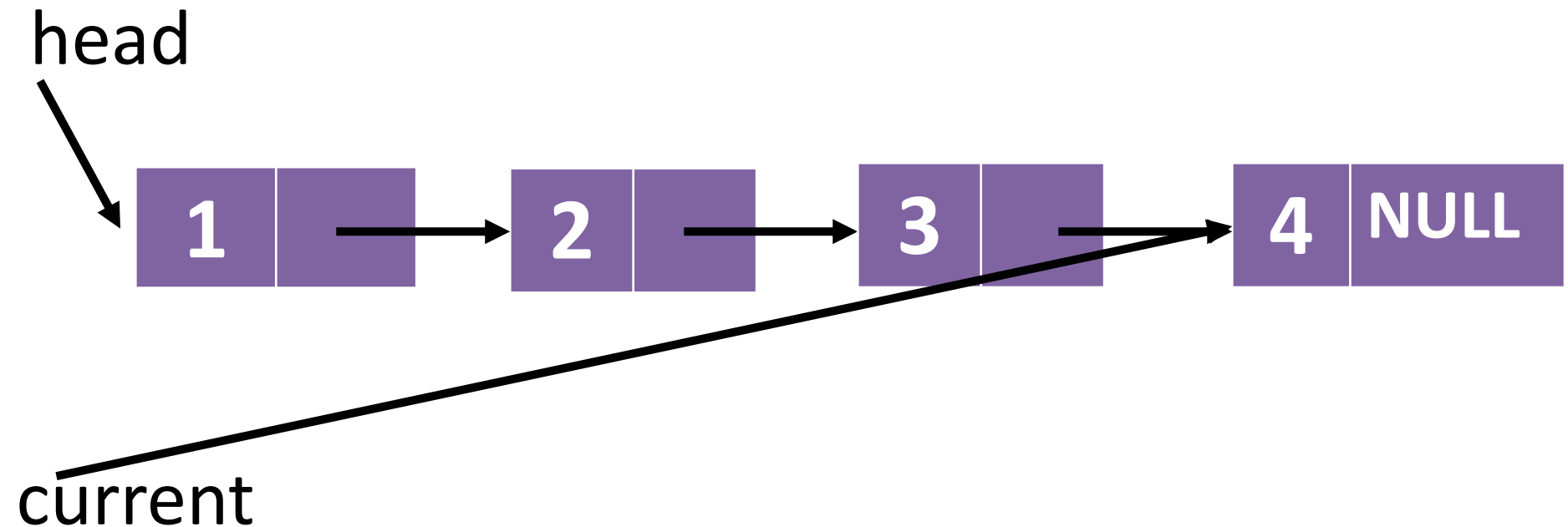
head

NULL

Deleting a node at end of list

head

Deleting a node at end of list

head

1 → 2 → 3 → 4 NULL

current

current->next == NULL
??? how do we set 3's next?

Deleting a node at end of list

head

current

1   →   **2**   →   **3**   →   **4** **NULL**
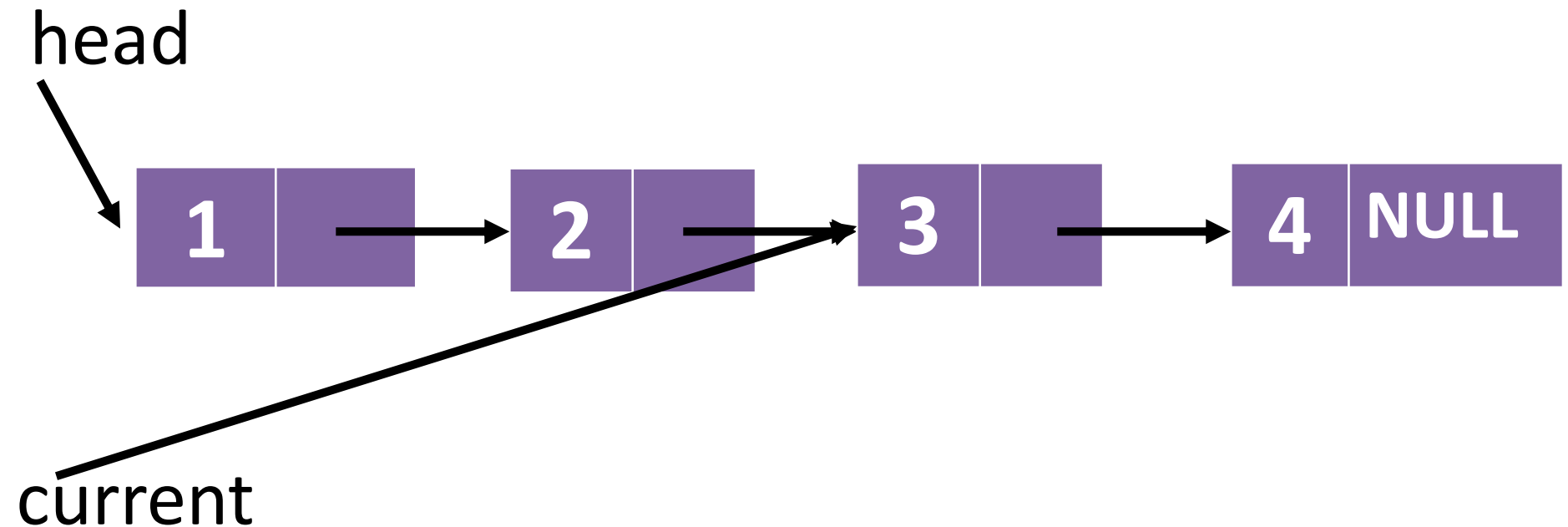
current->next->next == NULL
current->next = NULL