

Pointers and Dynamic Data Structures Chapter 13

Problem Solving & Program Design in C

Eighth Edition

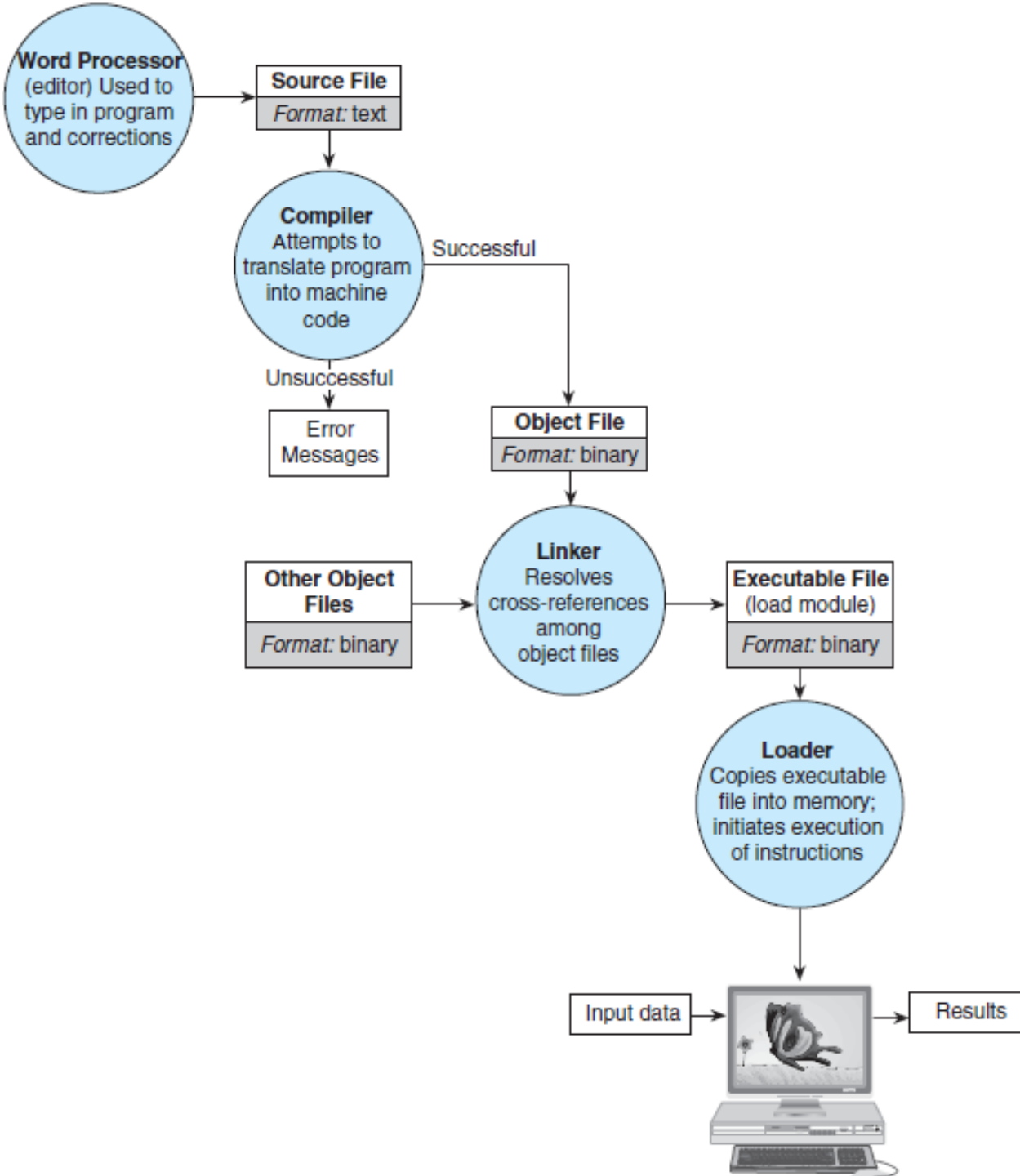
Jeri R. Hanly & Elliot B. Koffman

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



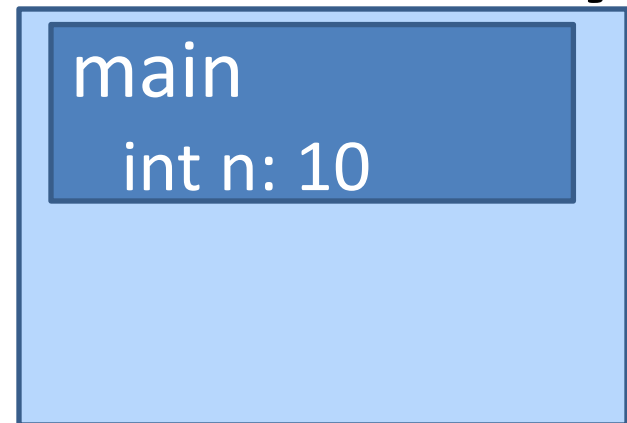
What happens when we run our executable file?



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

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

Stack memory



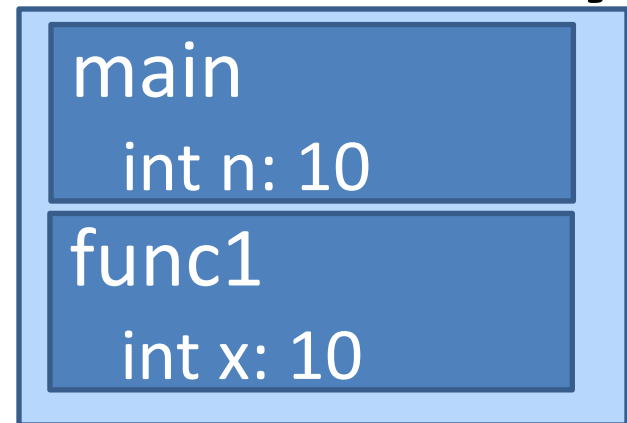
What happens when we run our executable file?



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

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

Stack memory



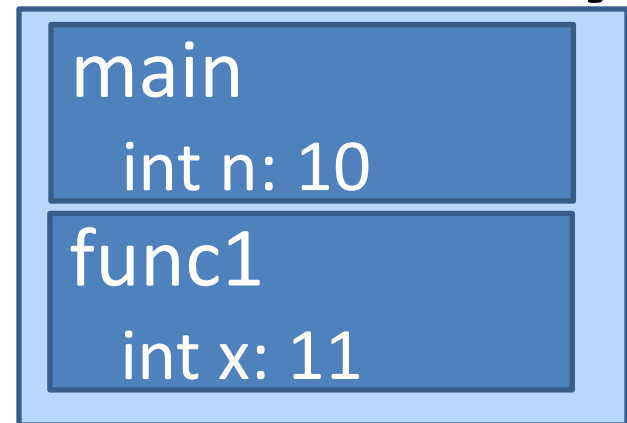
What happens when we run our executable file?



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

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

Stack memory



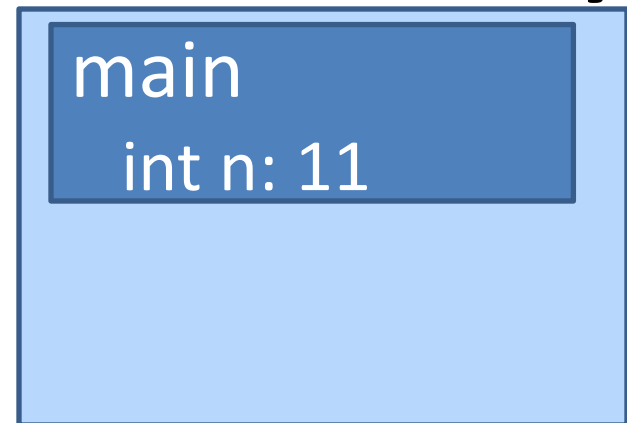
What happens when we run our executable file?



```
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

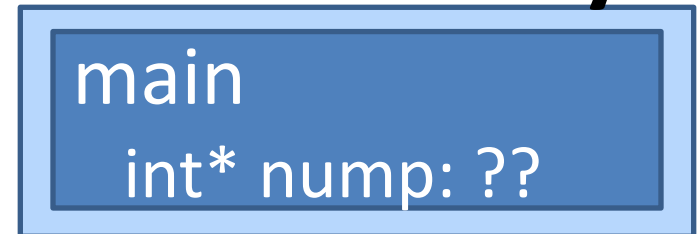
Heap memory

What happens when we run our executable file?



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

Stack memory



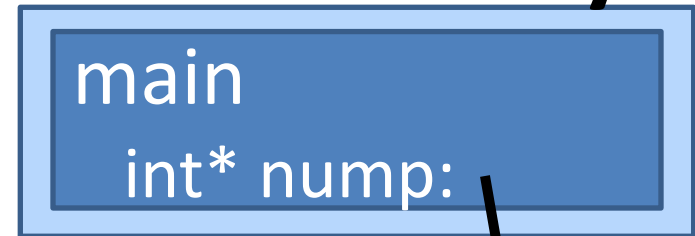
Heap memory

What happens when we run our executable file?



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

Stack memory



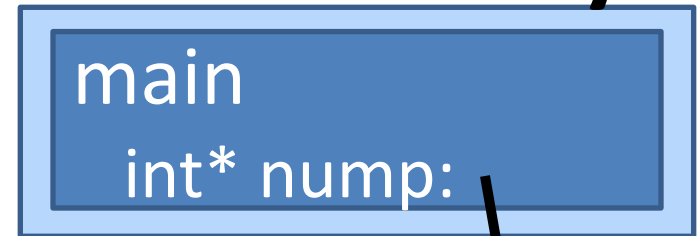
Heap memory

What happens when we run our executable file?



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

Stack memory



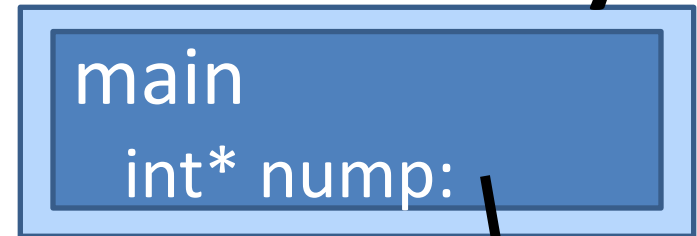
Heap memory

What happens when we run our executable file?



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

Stack memory



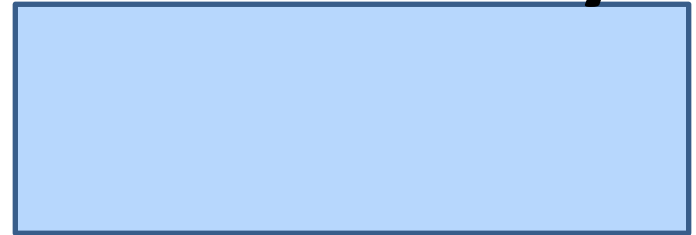
Heap memory

What happens when we run our executable file?



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

Stack memory



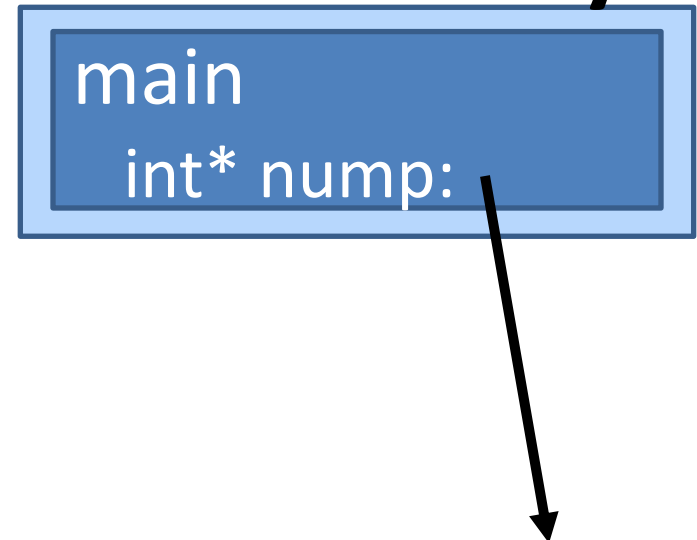
Heap memory

What happens when we run our executable file?



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

Stack memory



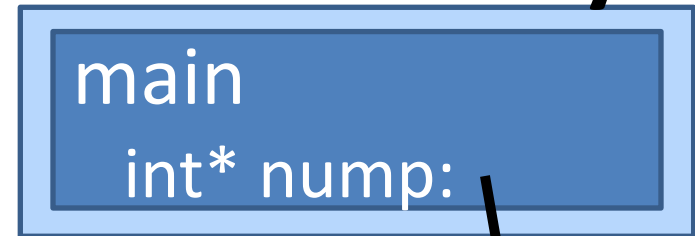
Heap memory

What happens when we run our executable file?



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

Stack memory



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?



```
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



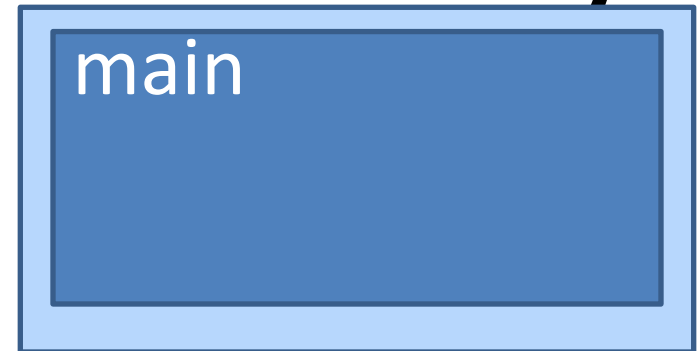
Heap memory

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



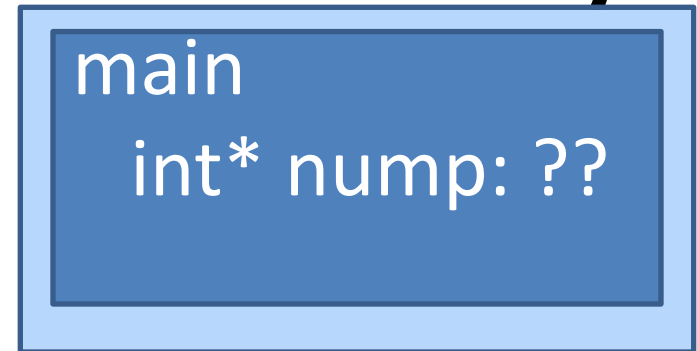
Heap memory

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



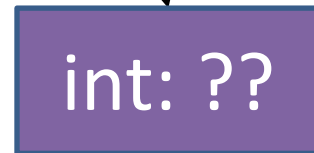
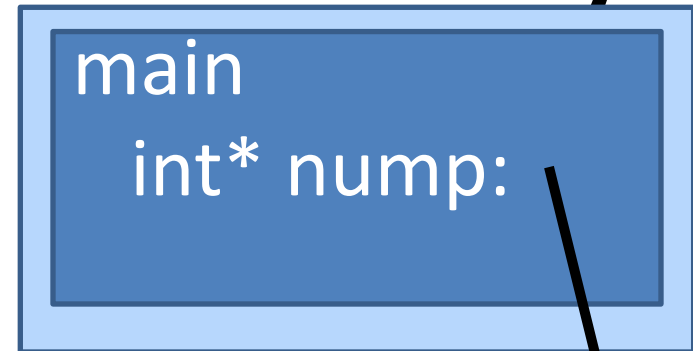
Heap memory

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



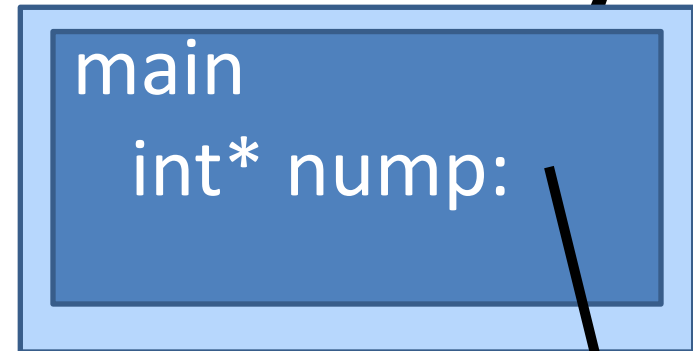
Heap memory

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

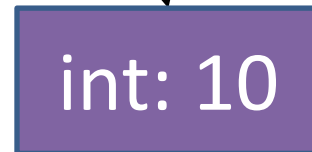
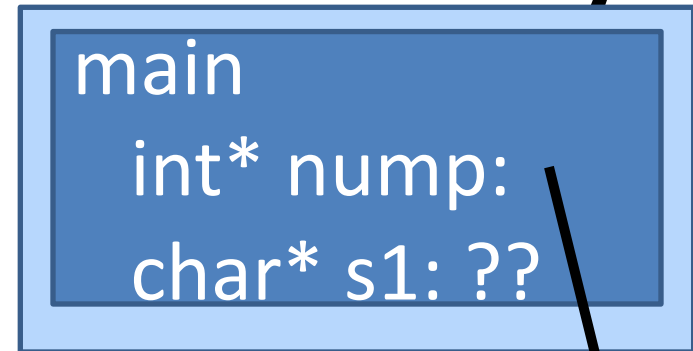
Heap memory

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



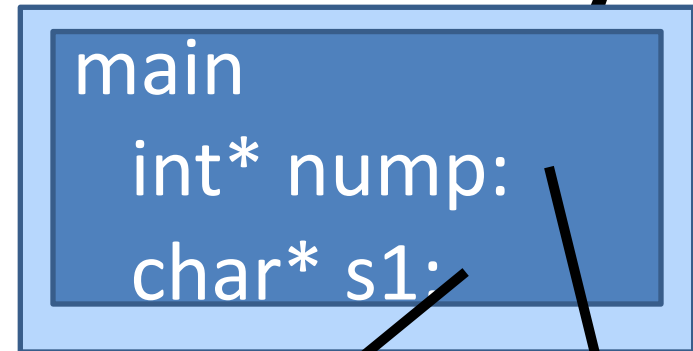
Heap memory

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



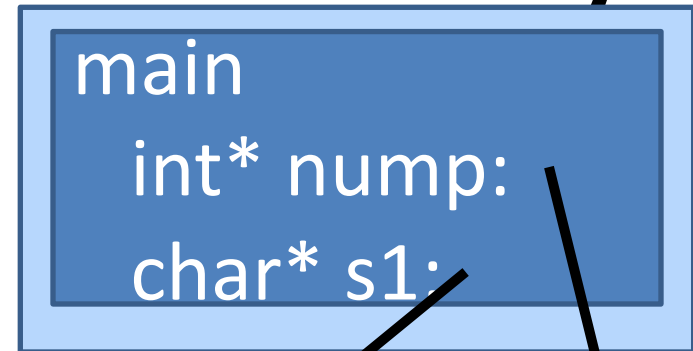
Heap memory

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	?	?	?	?
---	---	---	---	---	----	---	---	---	---

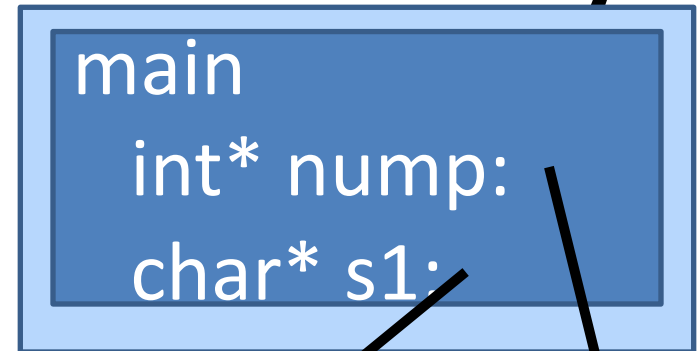
Heap memory

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



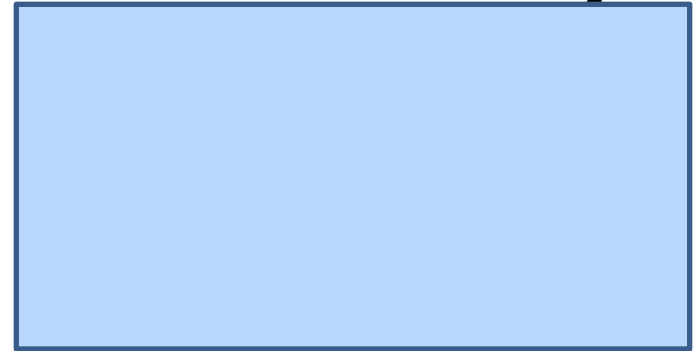
Heap memory

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	?	?	?	?
---	---	---	---	---	----	---	---	---	---

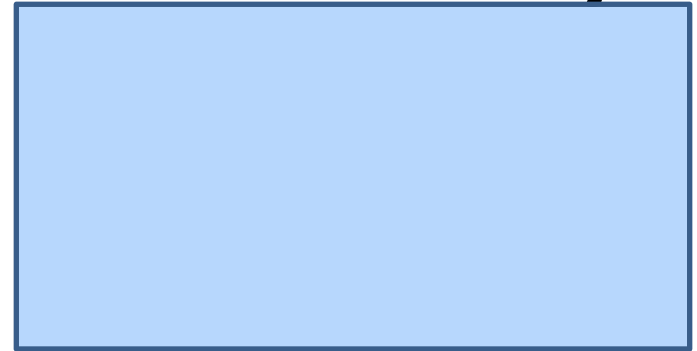
Heap memory

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!)

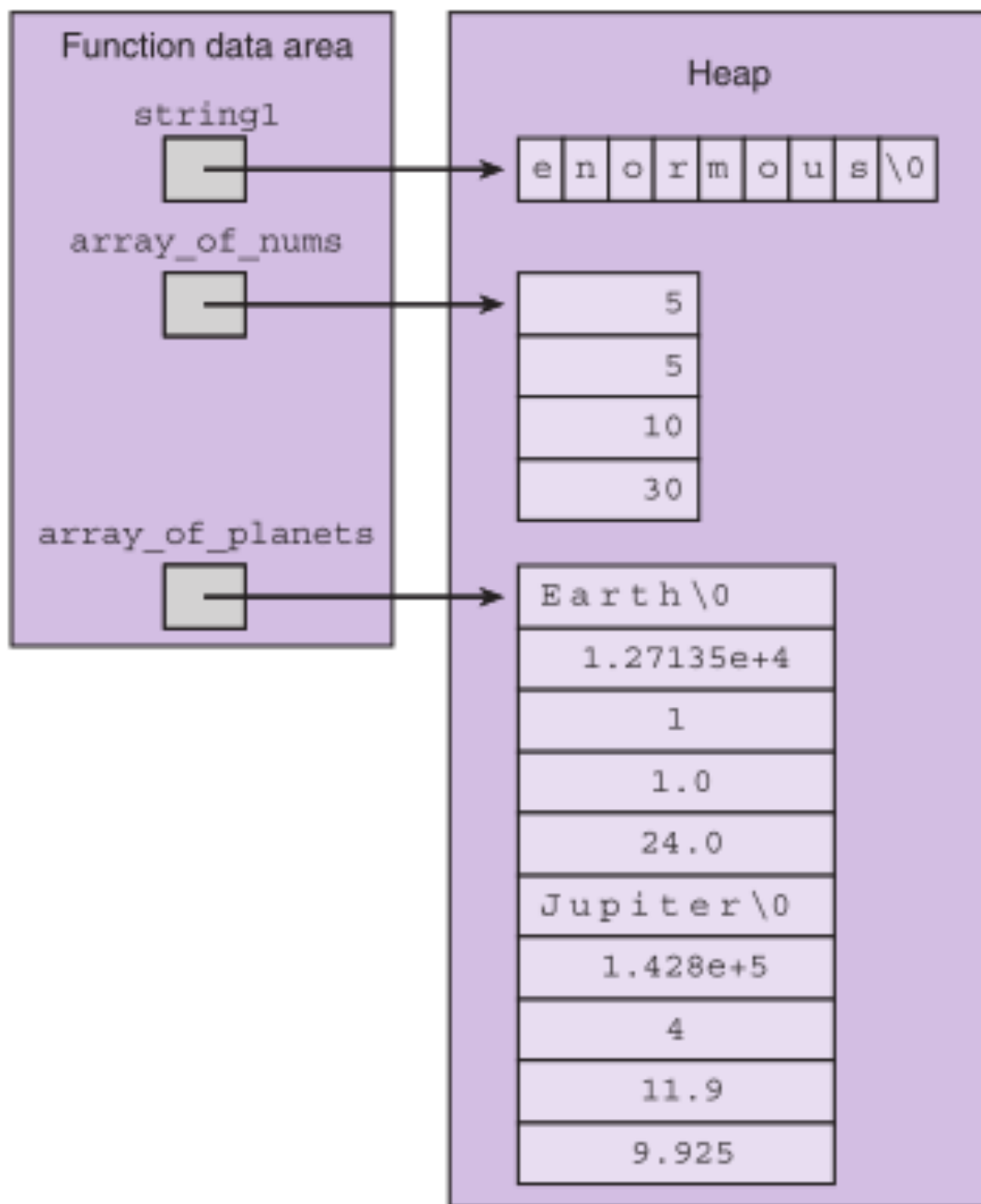
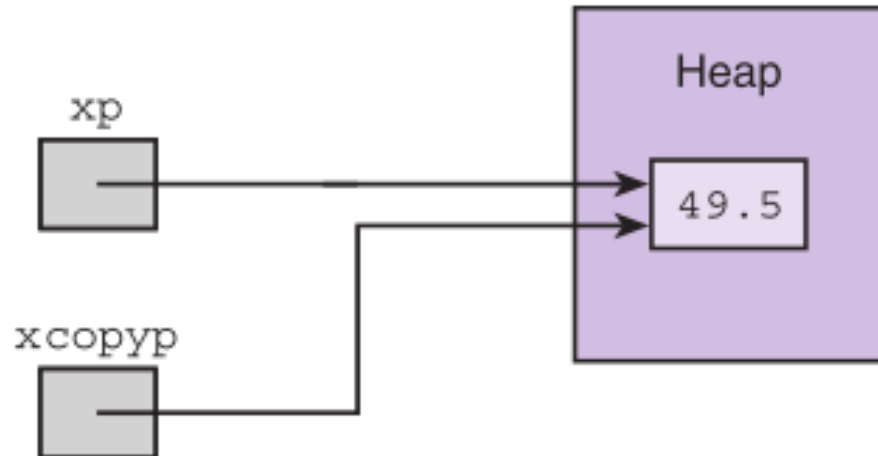


FIGURE 13.9

Multiple Pointers
to a Cell in the
Heap

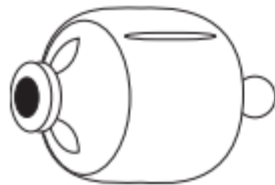


```
double *xp, *xcopy;  
  
xp = (double *)malloc(sizeof (double));  
*xp = 49.5;  
xcopy = xp;  
free(xp);  
. . .
```

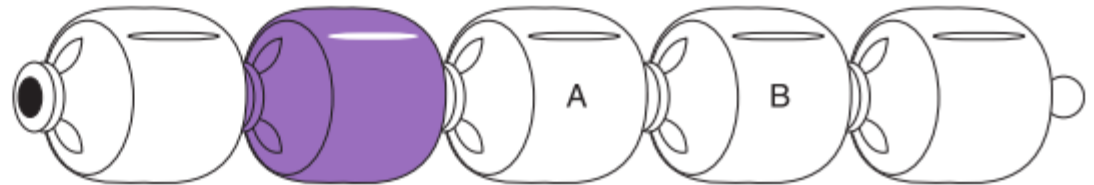
Linked Lists

- 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

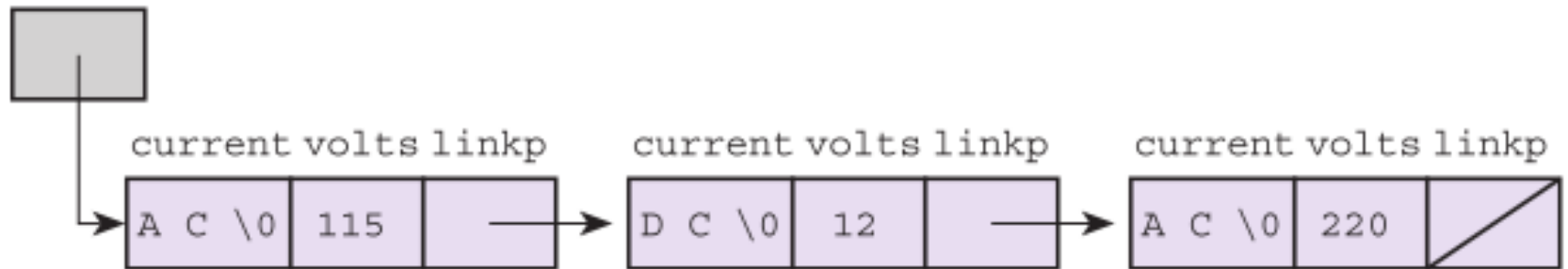
FIGURE 13.10 Children's Pop Beads in a Chain



Pop bead



Chain of pop beads



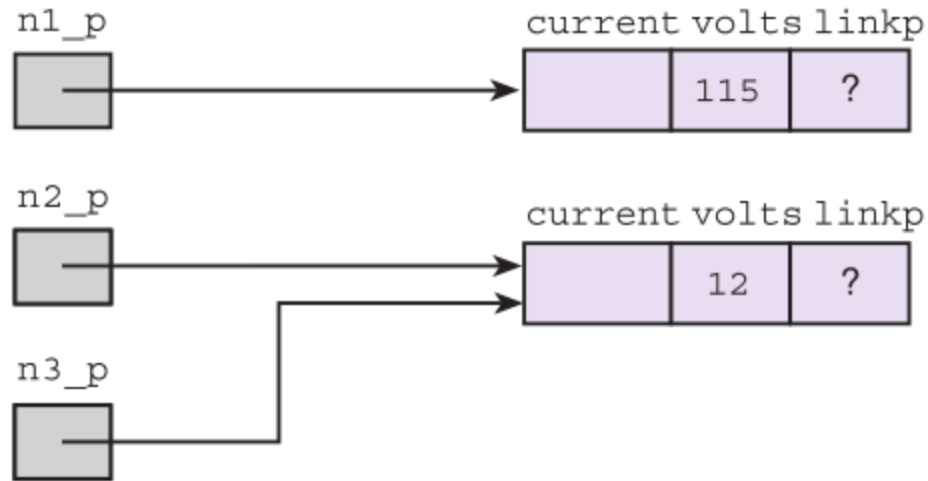


FIGURE 13.11

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

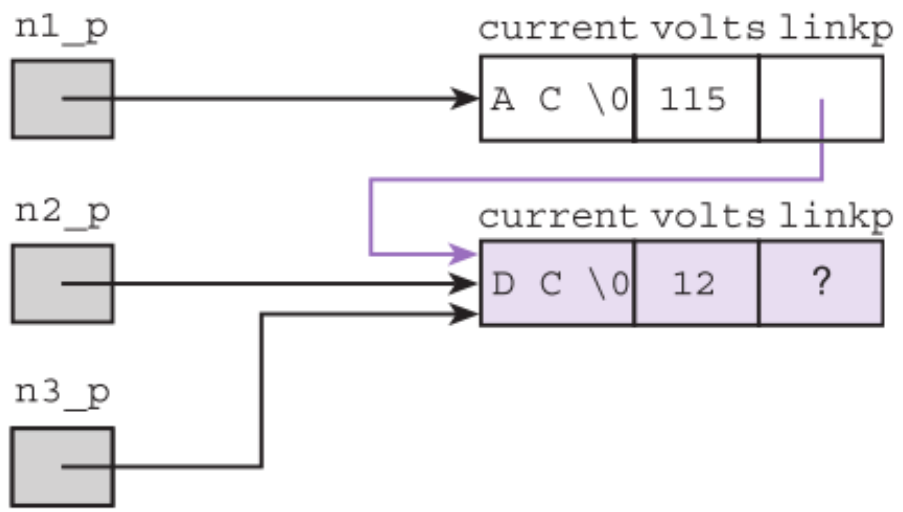
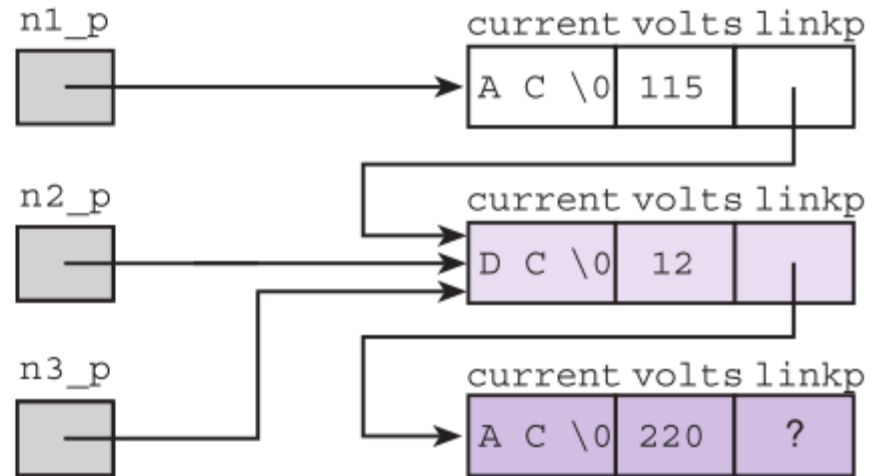


TABLE 13.2 Analyzing the Reference `n1_p->linkp->volts`

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



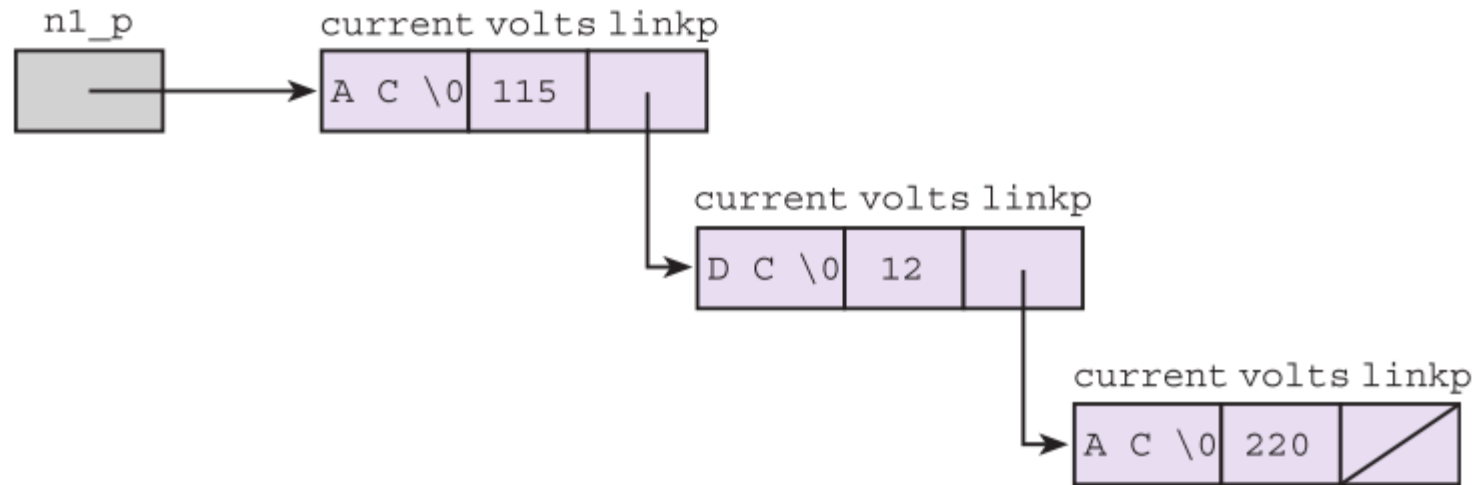


FIGURE 13.14

Three-Element
Linked List
Accessed Through
`n1_p`

```
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

```
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



The diagram illustrates the stack memory layout. It consists of a large light blue rectangle representing the stack. Inside the top portion of this rectangle is a smaller, darker blue rectangle labeled 'main'. This indicates that the 'main' function is currently executing in the stack memory.

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: ??

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

Heap memory

```

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

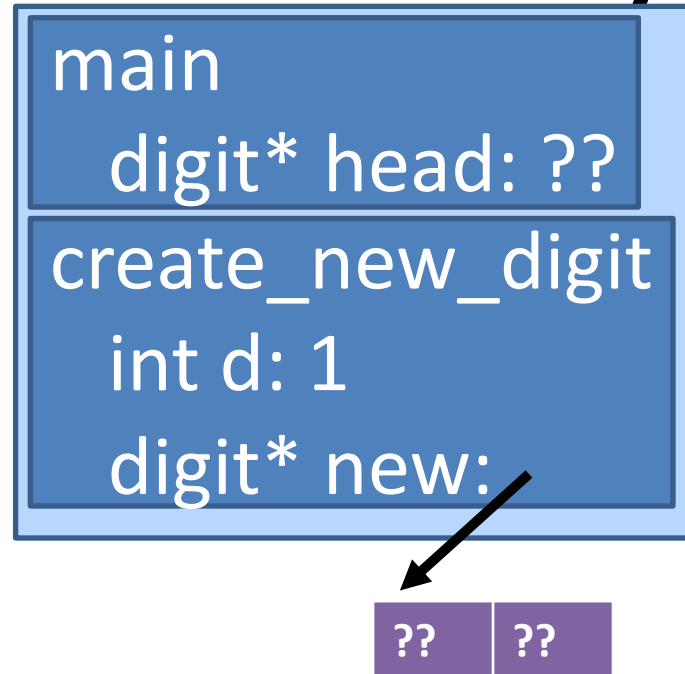
```

```

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



Heap memory


```

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

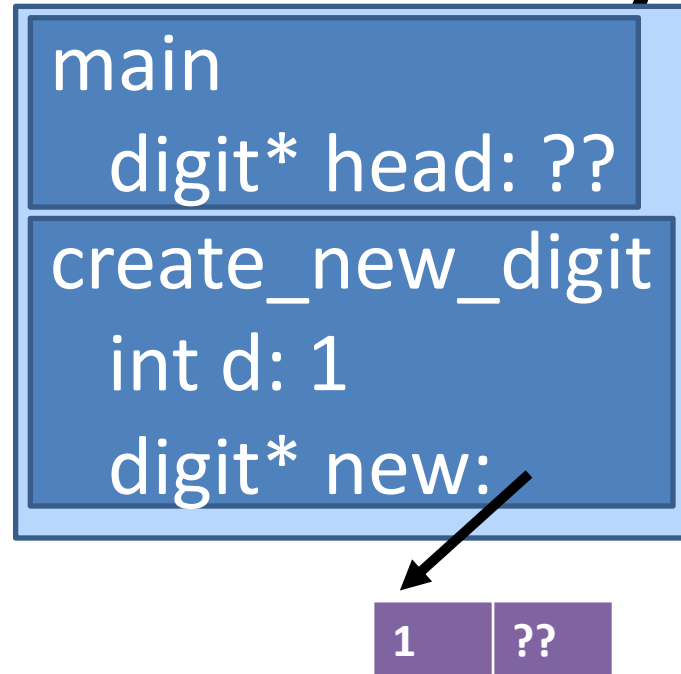
```

```

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



Heap memory

```

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

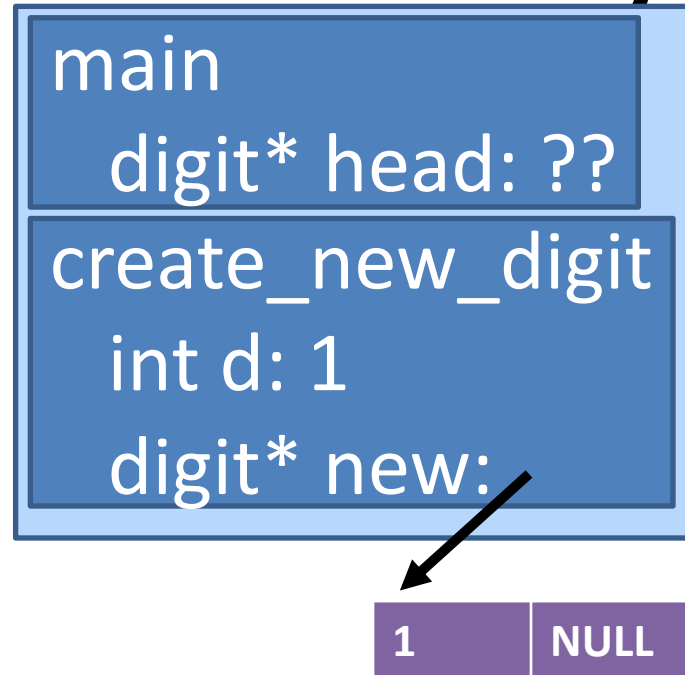
```

```

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



Heap memory

```

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

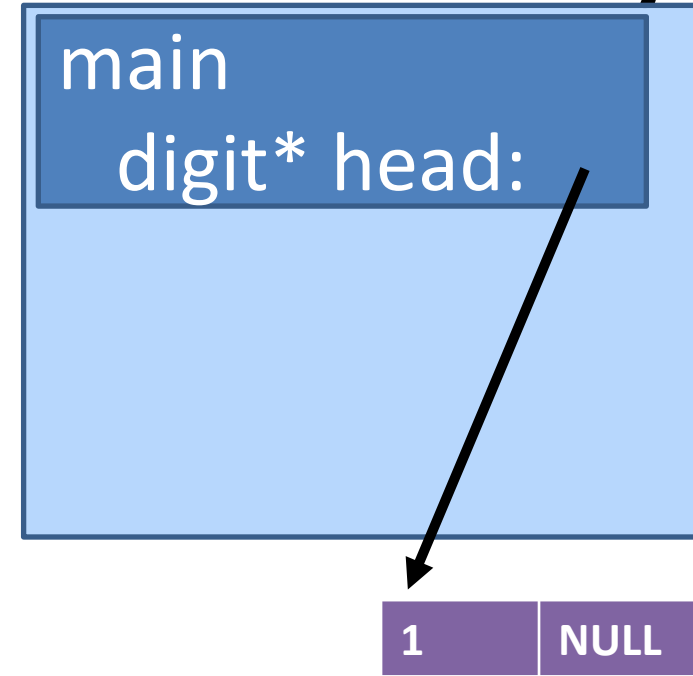
```

```

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



Heap memory

```

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

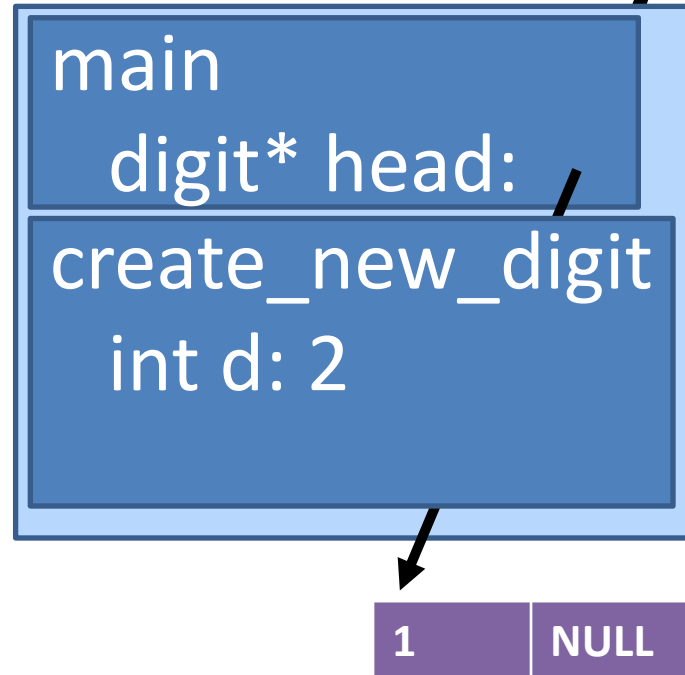
```

```

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



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



Heap memory

```

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

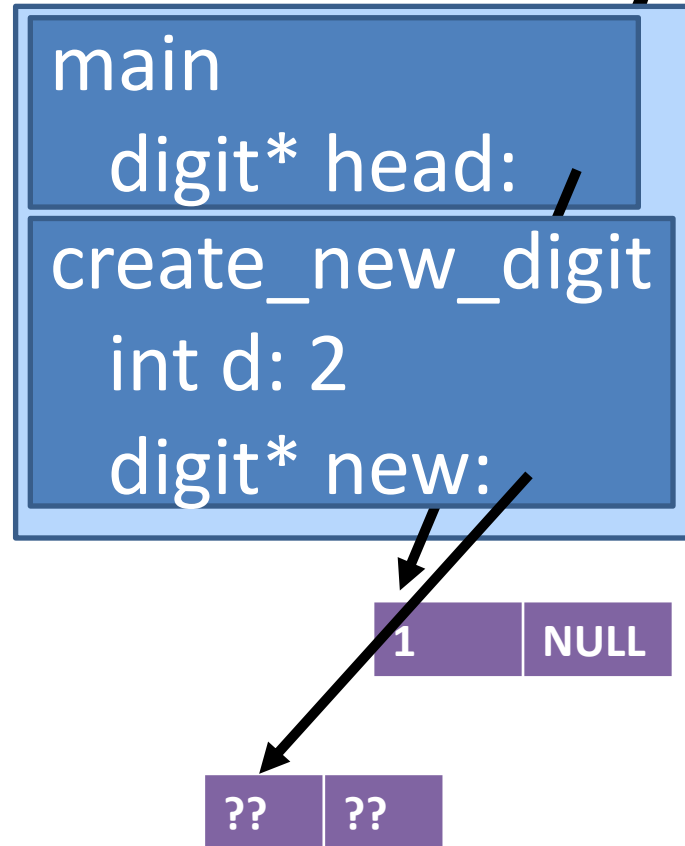
```

```

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



Heap memory

```

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

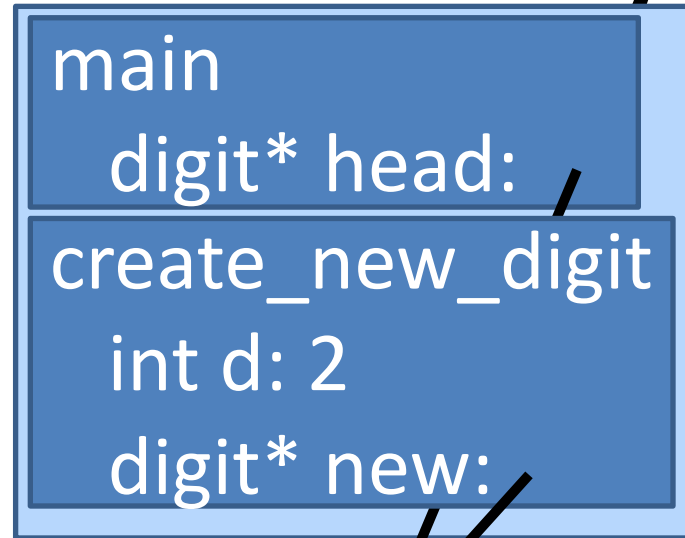
```

```

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



Heap memory

```

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

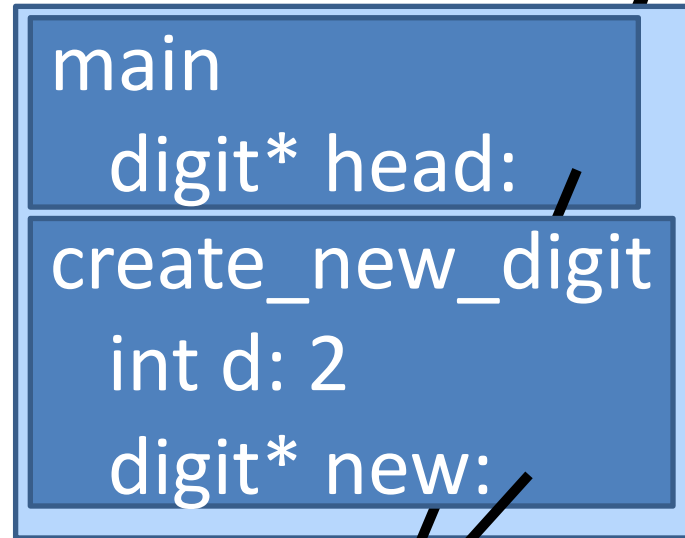
```

```

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



Heap memory


```

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

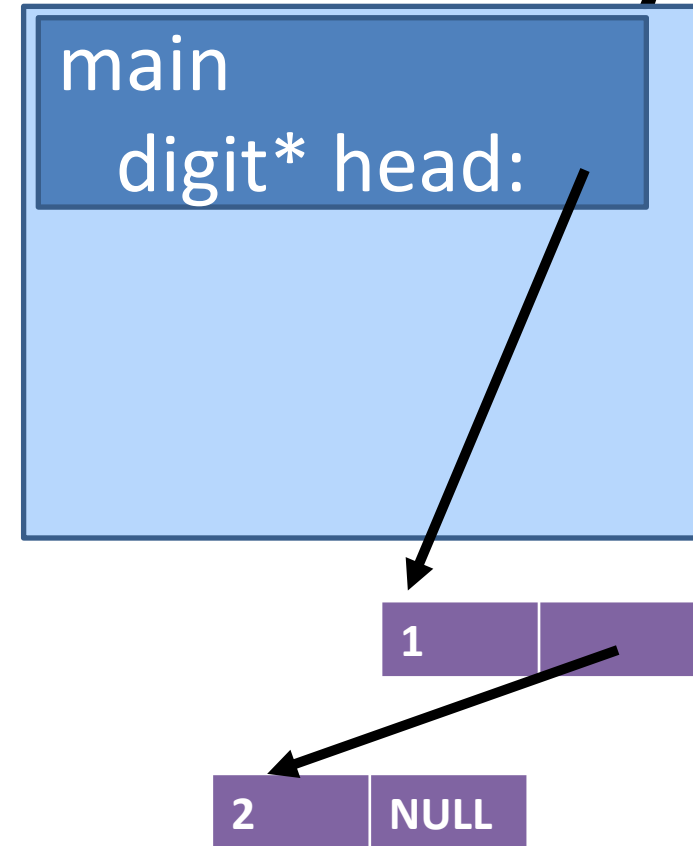
```

```

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



Heap memory

```

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

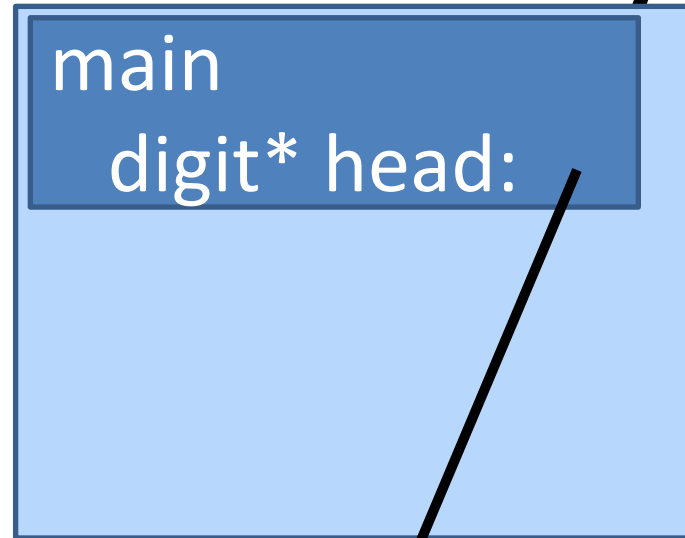
```

```

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



Heap memory

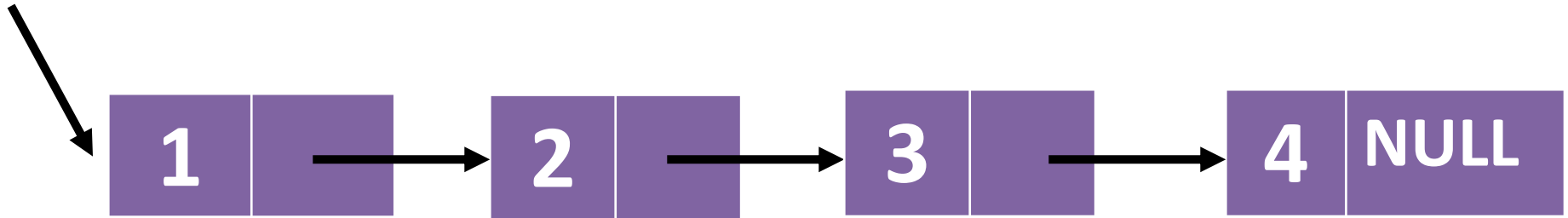
Inserting a node at end of list

head



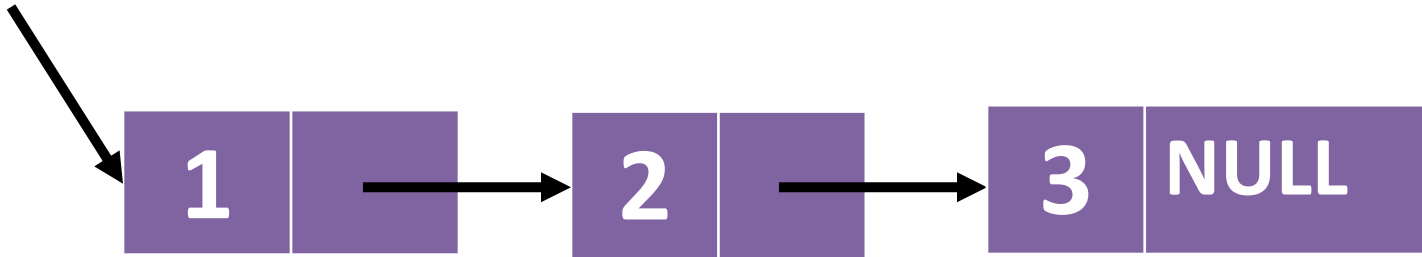
Inserting a node at end of list

head

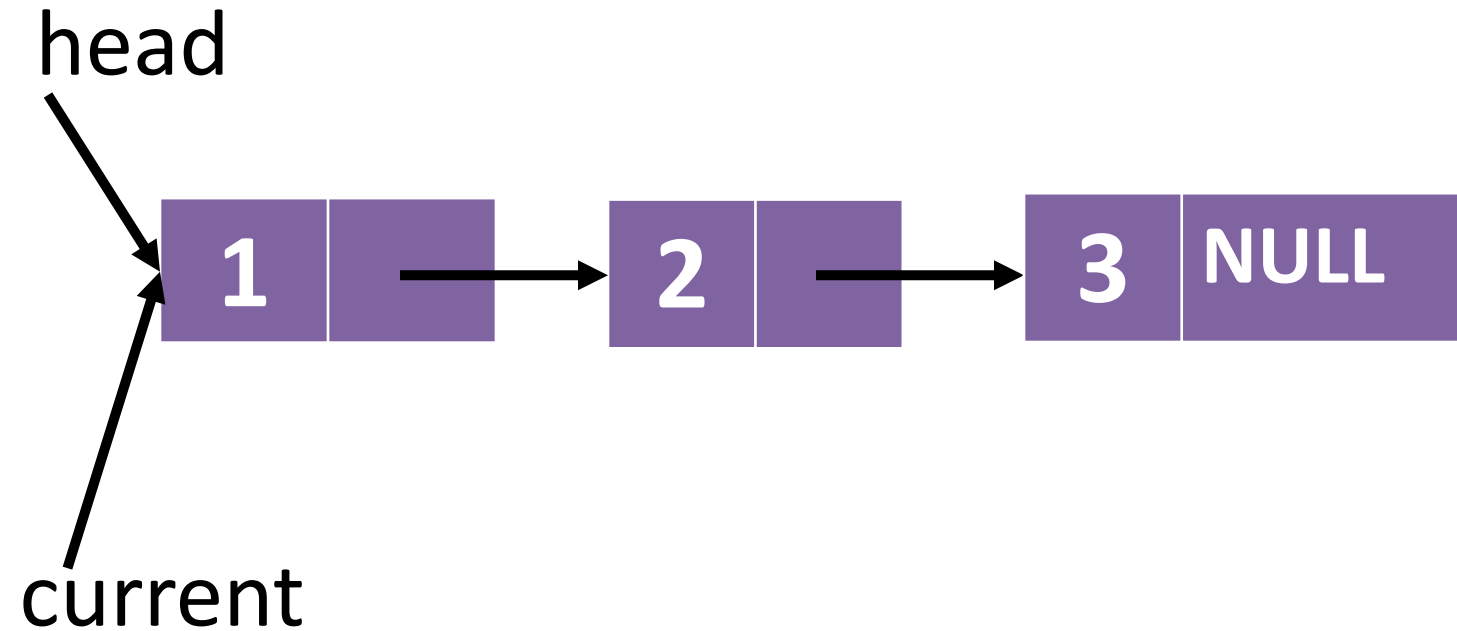


Inserting a node at end of list

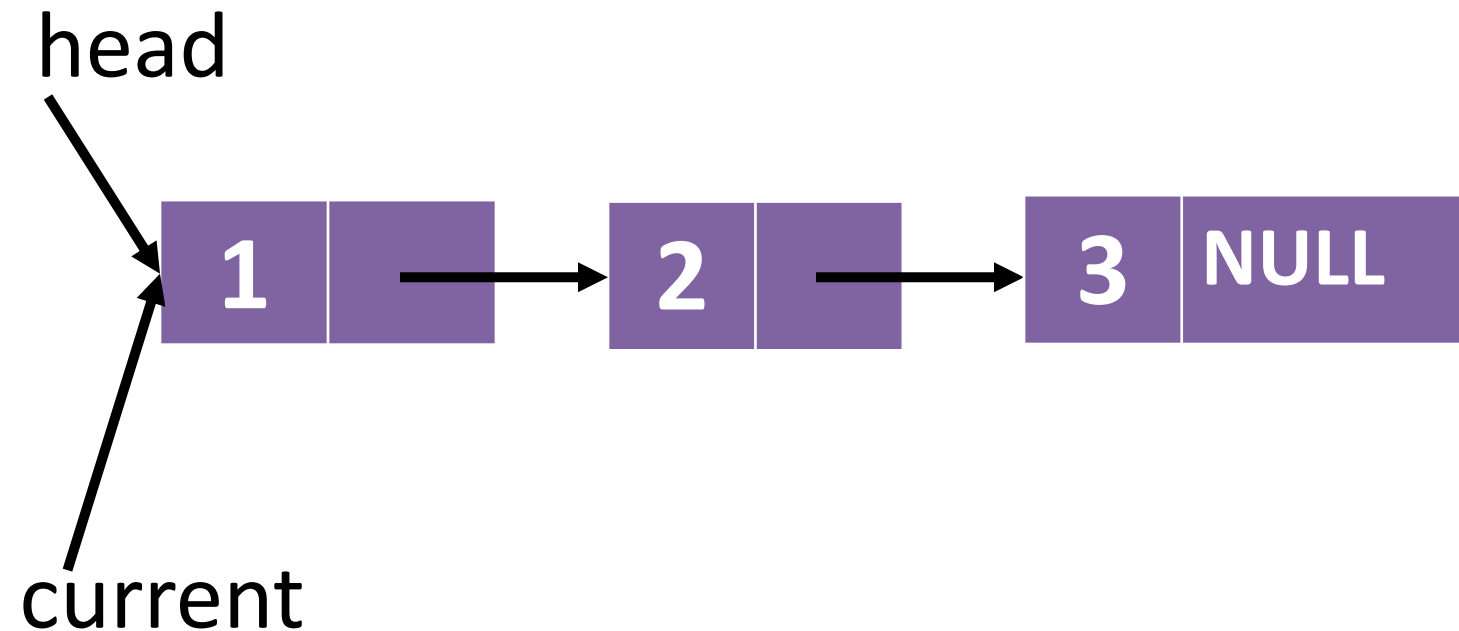
head



Inserting a node at end of list

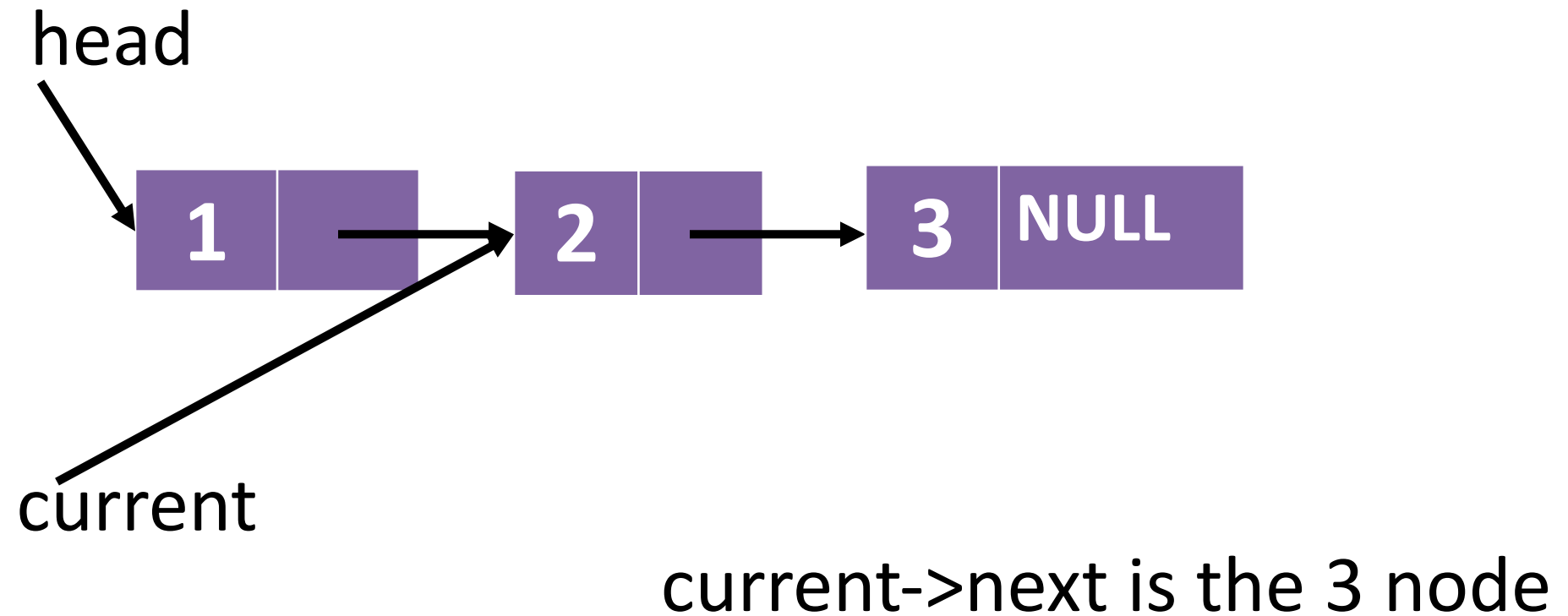


Inserting a node at end of list

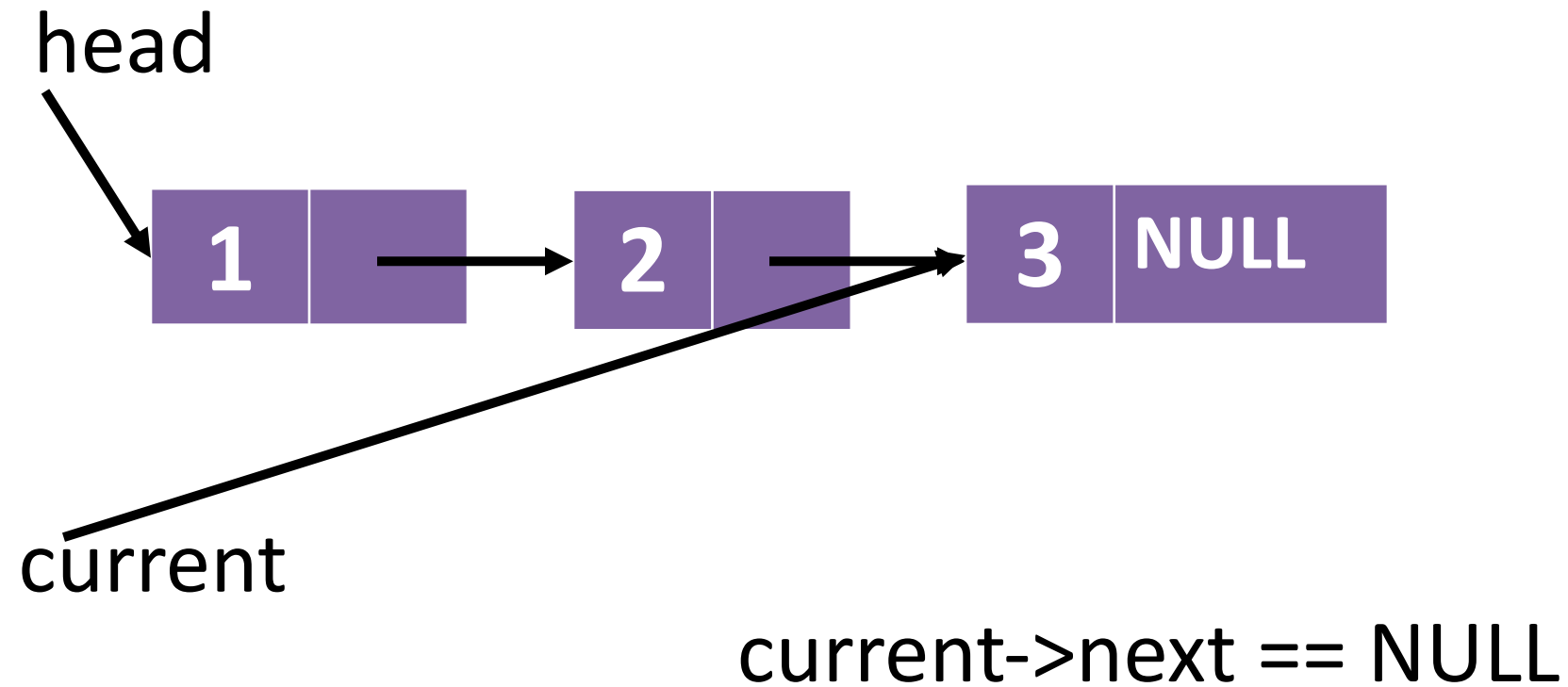


current->next is the 2 node

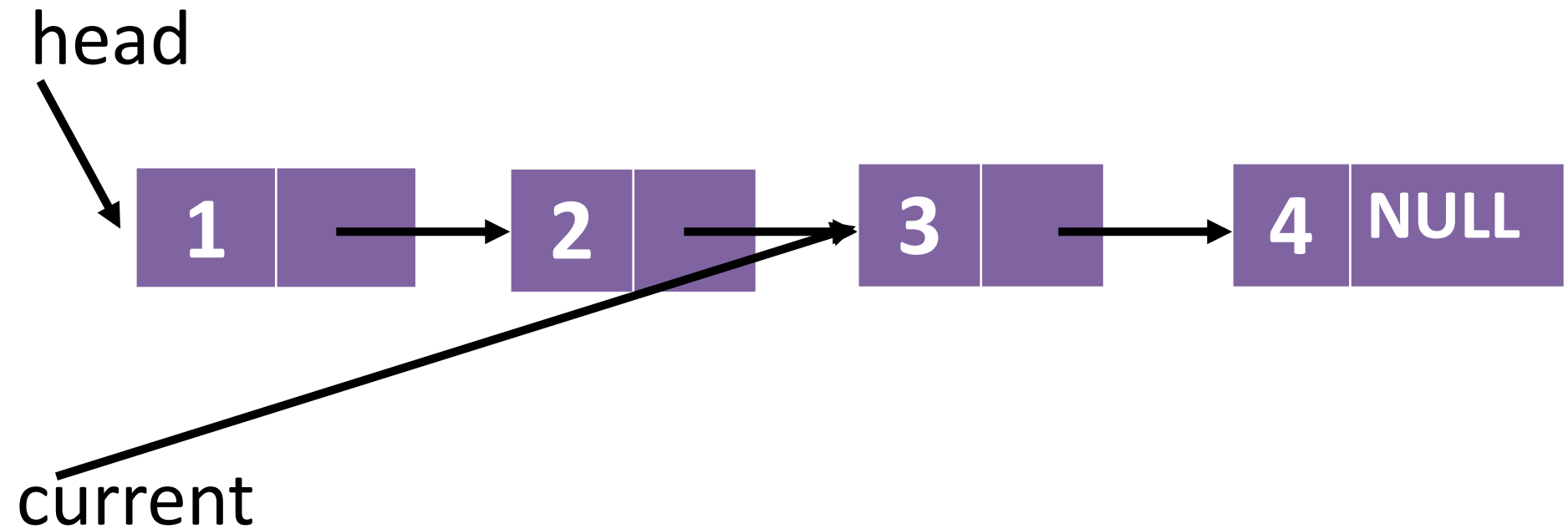
Inserting a node at end of list



Inserting a node at end of list



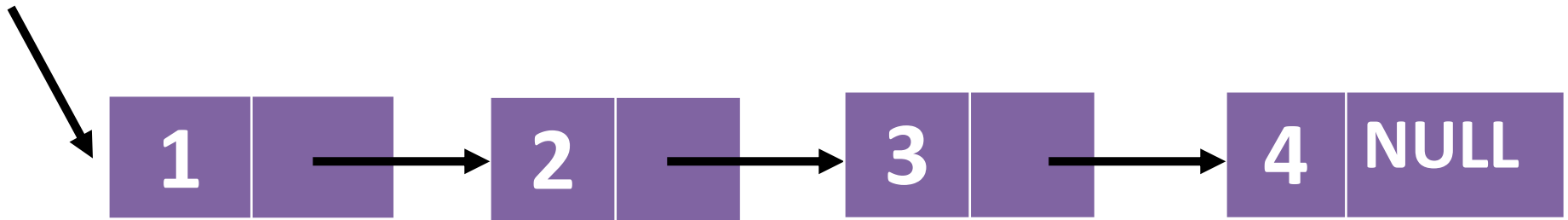
Inserting a node at end of list



`current->next = new`

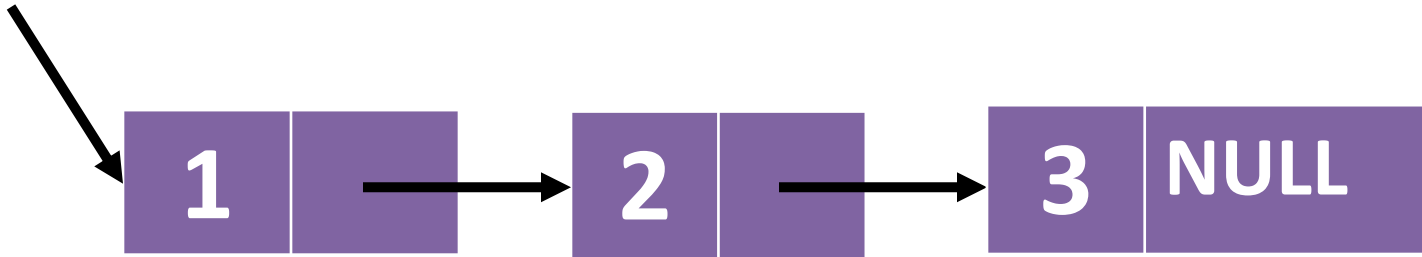
Deleting a node at end of list

head

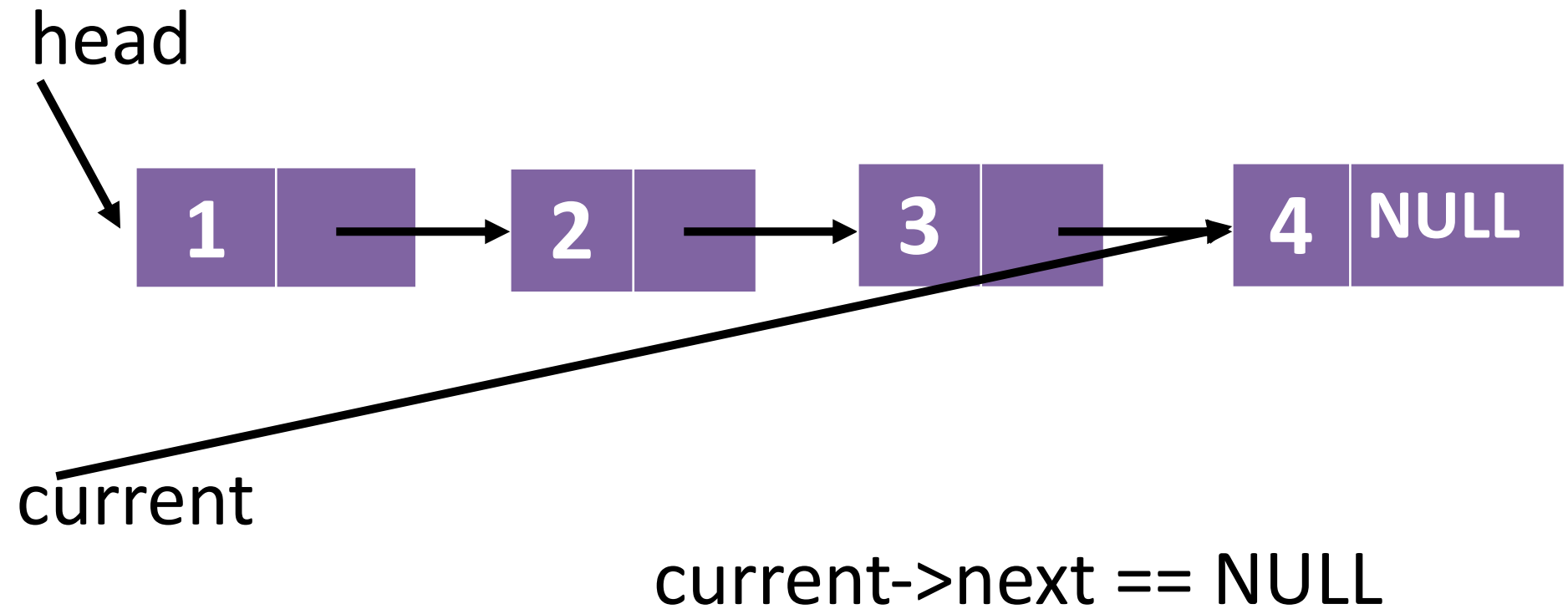


Deleting a node at end of list

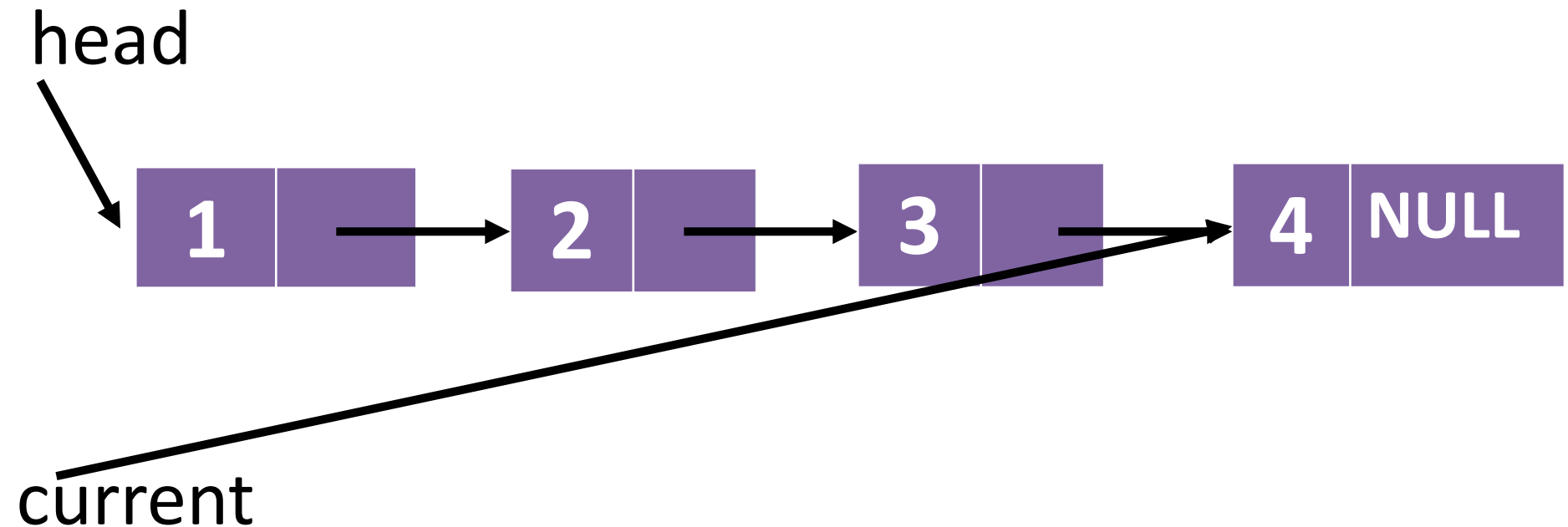
head



Deleting a node at end of list



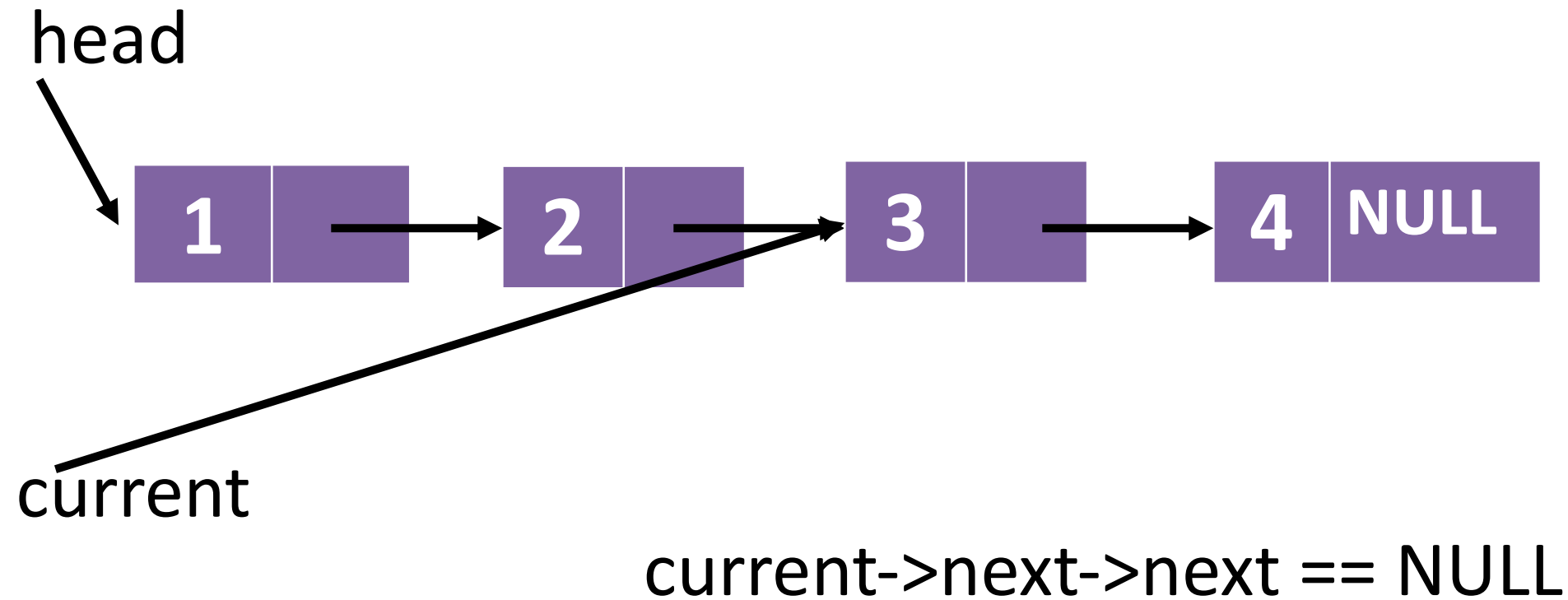
Deleting a node at end of list



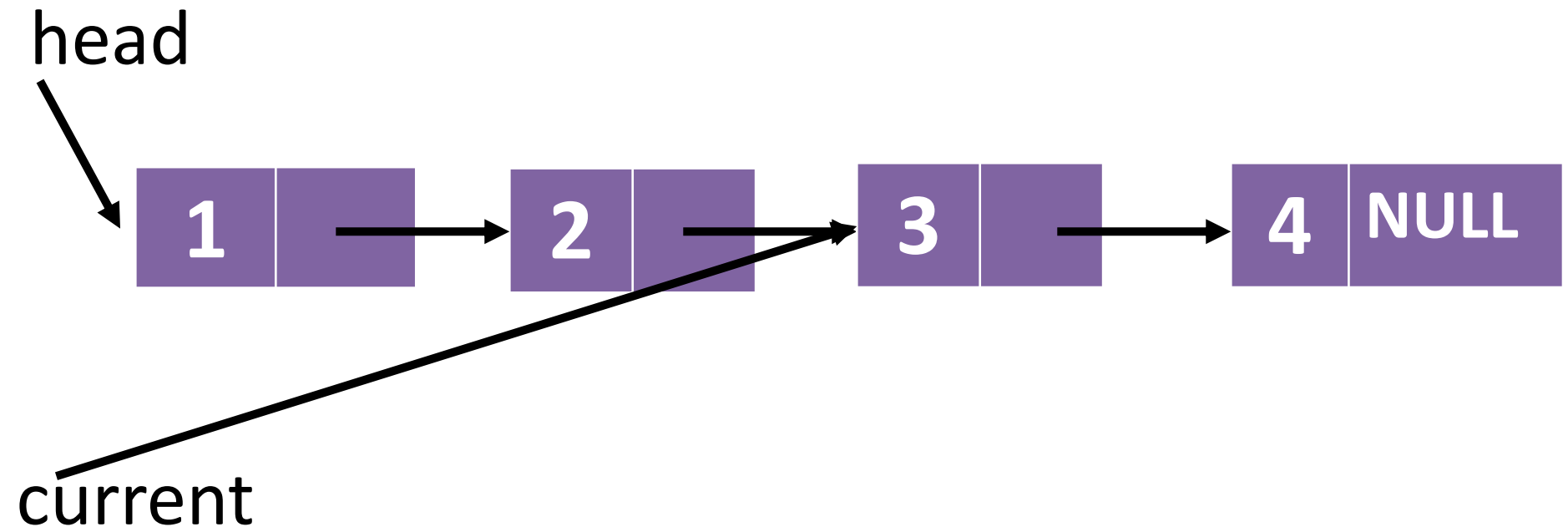
`current->next == NULL`

??? how do we set 3's next?

Deleting a node at end of list

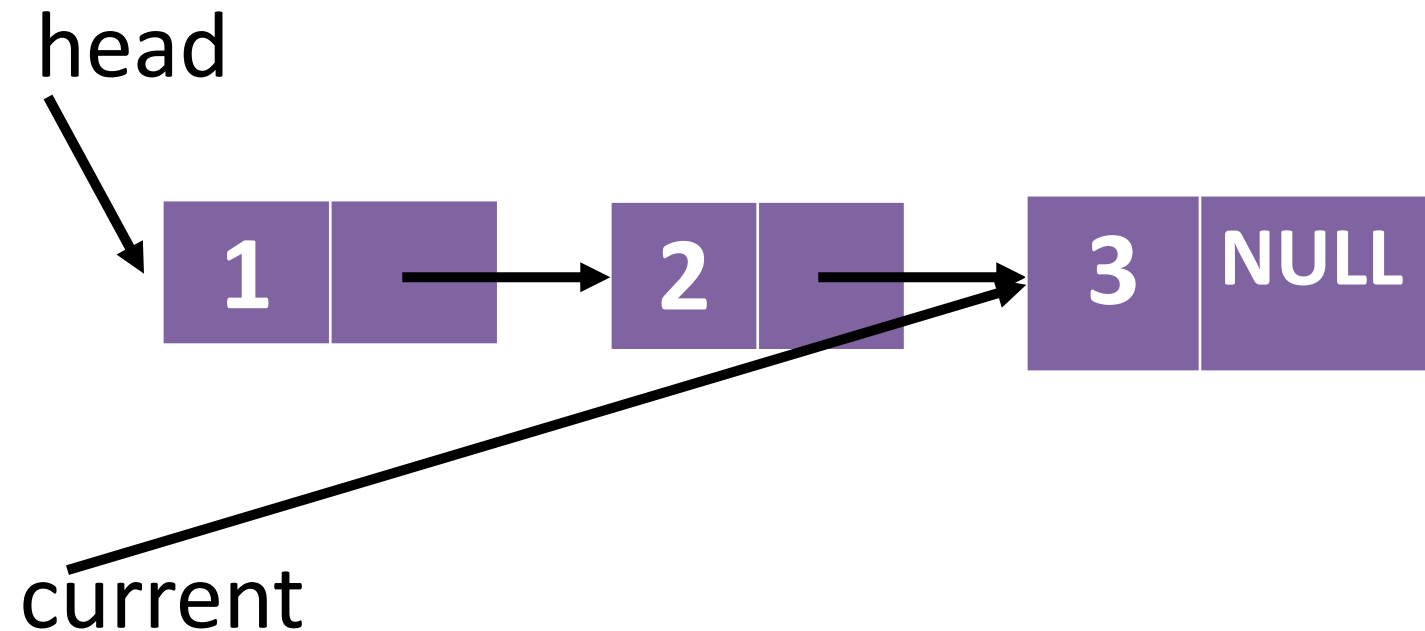


Deleting a node at end of list



```
current->next->next == NULL  
current->next = NULL
```


Deleting a node at end of list



```
current->next->next == NULL  
current->next = NULL
```

Deleting a node at end of list

head



What happens when we do

current->next->next == NULL

?

Appendix A: More about pointers

Pointer arithmetic:

$p+1$

moves p forward to the next memory block
of `sizeof(*p)`

A dynamic array in C?

- We can use realloc

realloc(<pointer to allocated memory>, <size>)