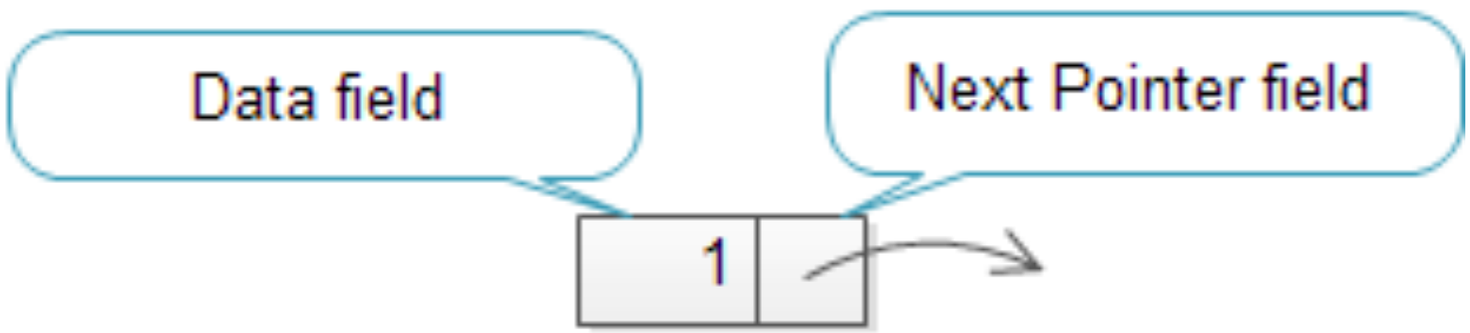
A C Primer (8):
Linked Lists, Enums, and Function Pointers

Ion Mandoiu
Laurent Michel
Revised by M. Khan, J. Shi and W. Wei

Example: Linked list

```
// A data structure that consists of a chain of nodes
// Starting from head, a node has a reference to the next node

typedef struct node_tag {
   int    v;                    // data
   struct node_tag * next;  // A pointer to this type of struct
} node;                         // Define a type. Easier to use.
```

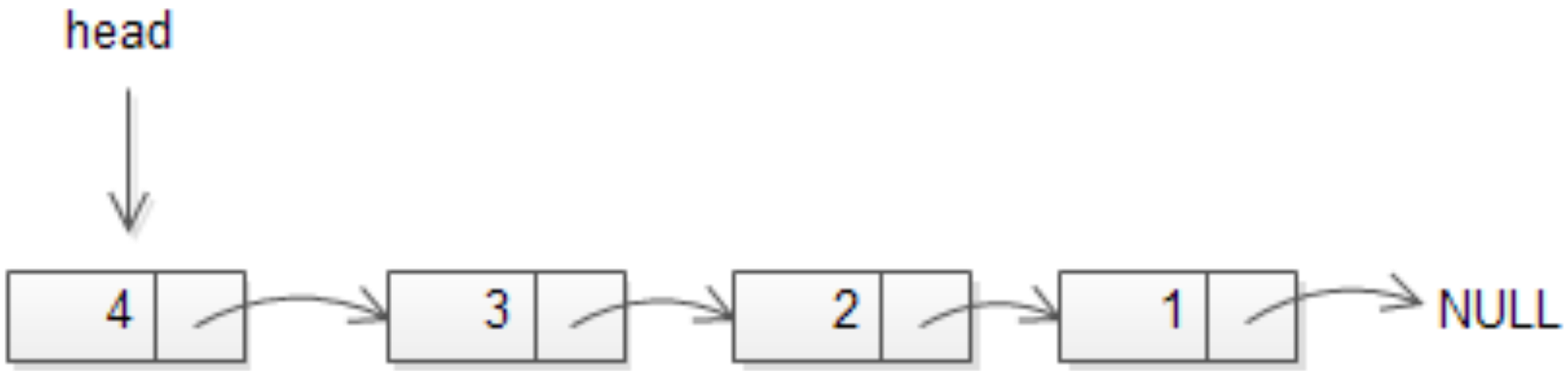


Figures for linked list are from http://www.zentut.com/c-tutorial/c-linked-list/

Head

```
node * head;  // head is a pointer, not a node!

head = NULL;  // at beginning, it is empty
```

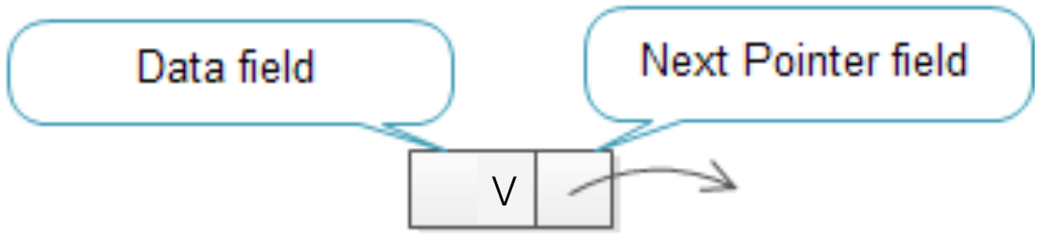
After adding nodes into the list,



Figures for linked list are from http://www.zentut.com/c-tutorial/c-linked-list/

Create a node

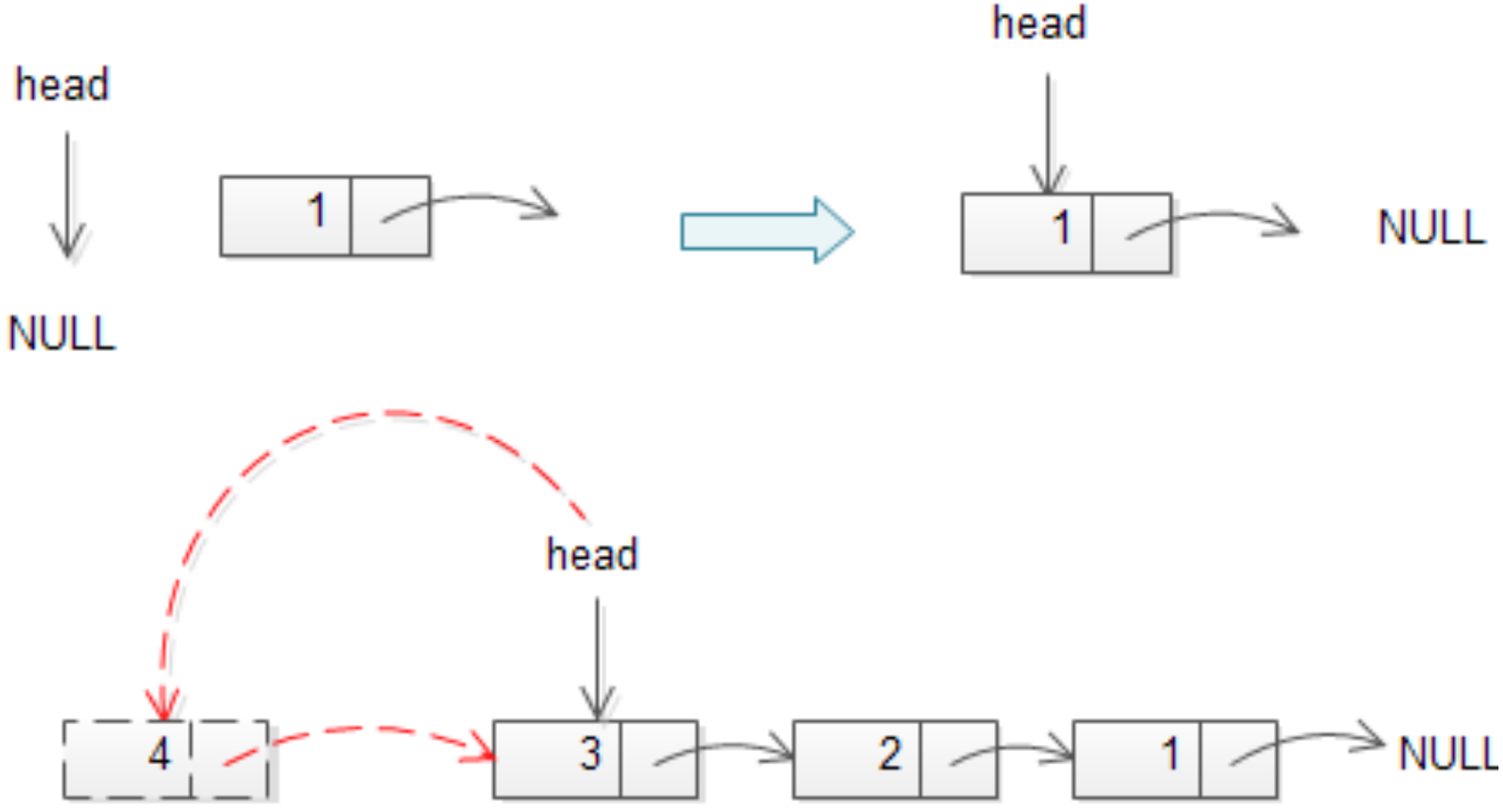
```
node * new_node(int v)      // create a node for value v
{
    node   * p = malloc(sizeof(node)); // Allocate memory
    assert(p != NULL);       // you can be nicer

    // Set the value in the node.
    p->v = v;           // you could do (*p).v
    p->next = NULL;
    return p;           // return
}
// is it similar to creating objects using “new”?
```



Figures for linked list are from http://www.zentut.com/c-tutorial/c-linked-list/

Prepend

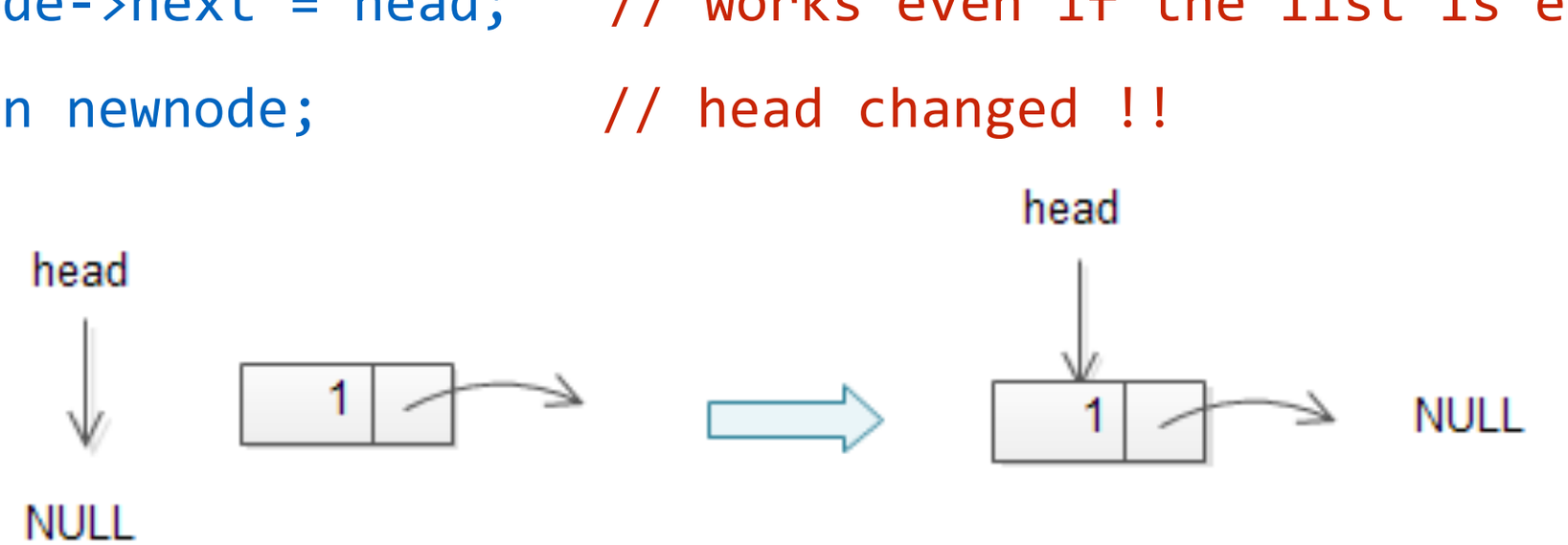
```
node * prepend(node * head, node * newnode)
{
    // How?
}
```

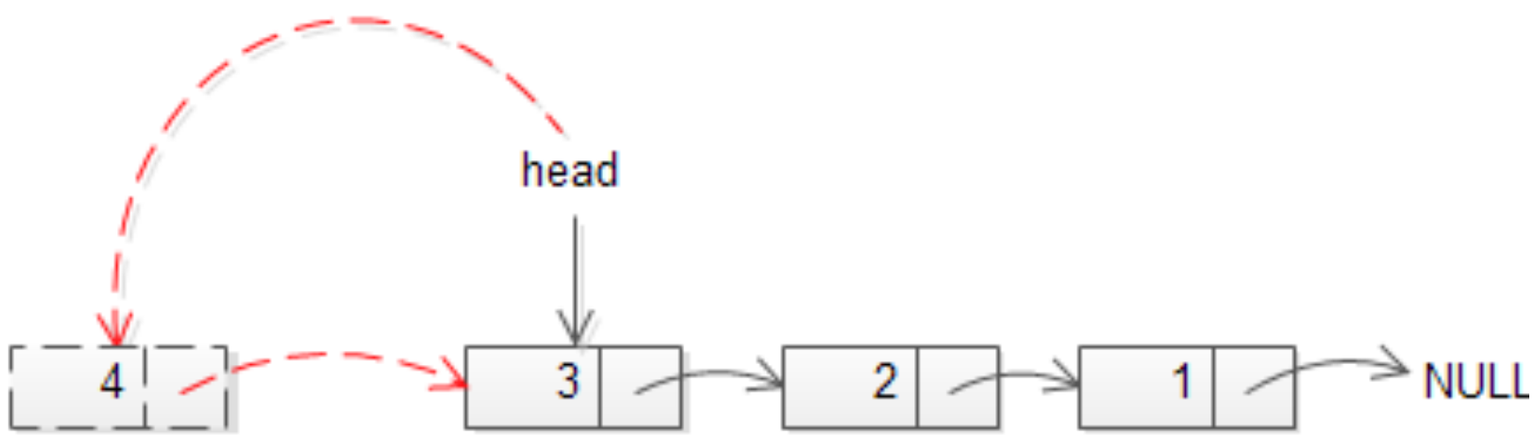


Figures for linked list are from http://www.zentut.com/c-tutorial/c-linked-list/

Prepend

```
node * prepend(node * head, node * newnode)
{
    newnode->next = head;    // works even if the list is empty
    return newnode;          // head changed !!
}
```

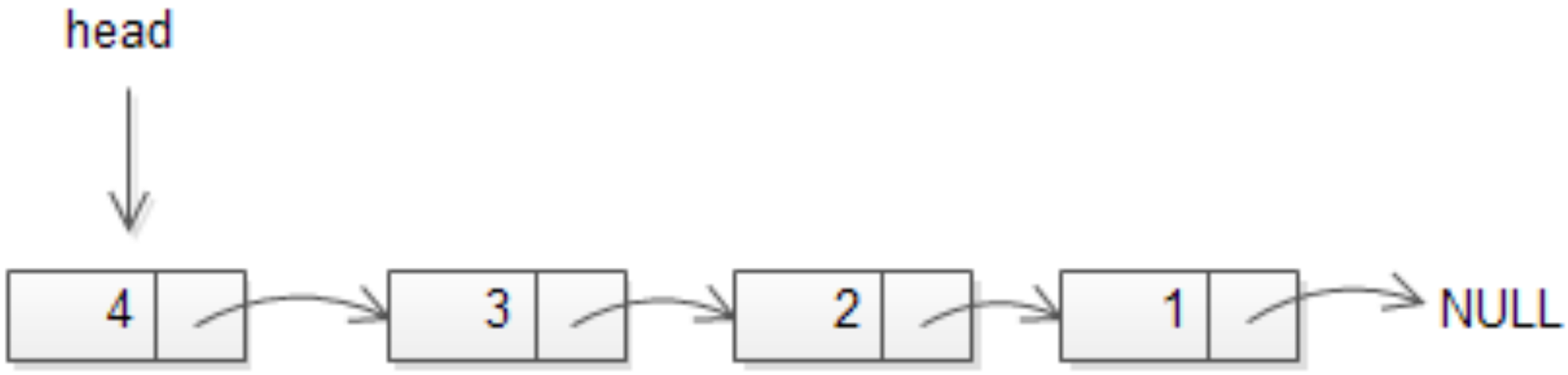




Figures for linked list are from http://www.zentut.com/c-tutorial/c-linked-list/

Find the last one

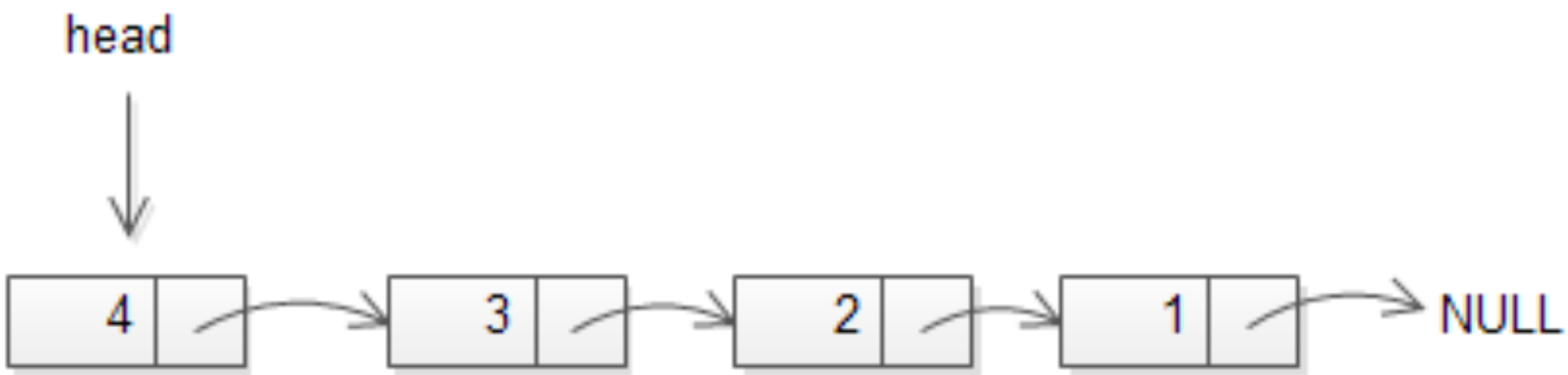
```
node * find_last(node * head)
{
    // How?
}
```



Figures for linked list are from http://www.zentut.com/c-tutorial/c-linked-list/

Find the last one

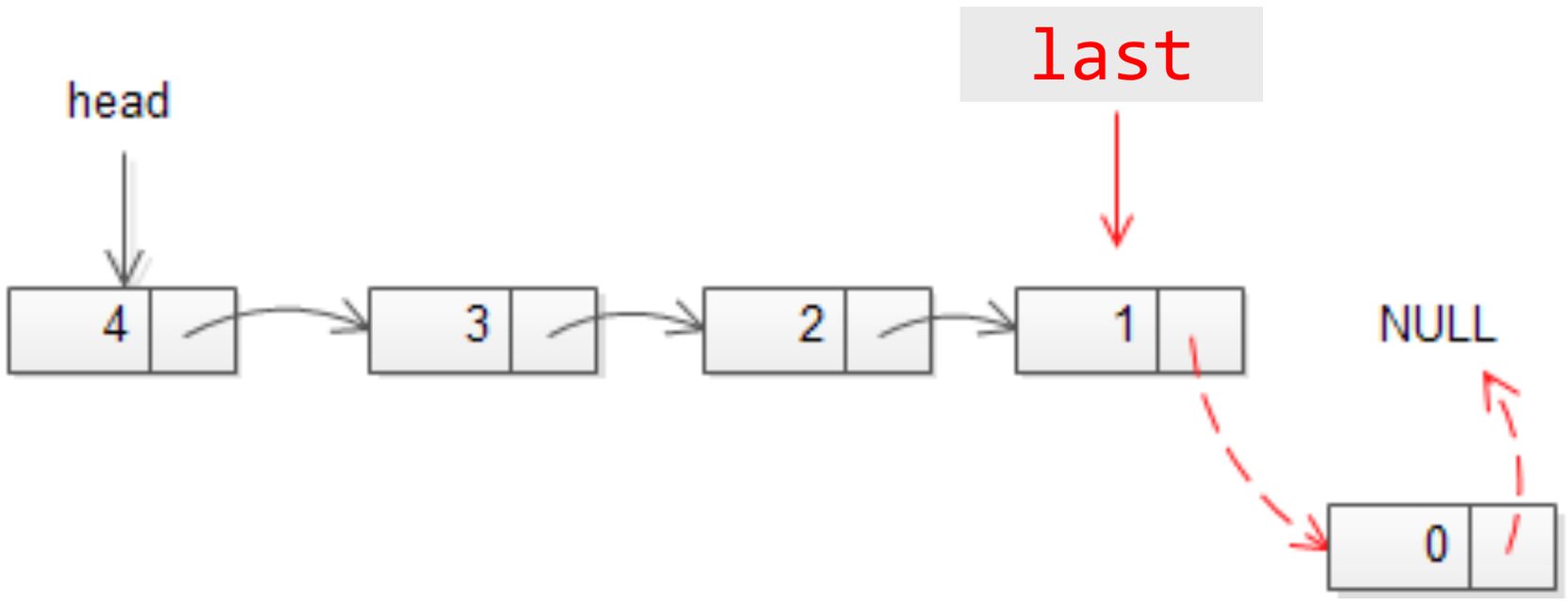
```
node * find_last(node * head)
{
    if (head != NULL) {          // only if the list is not empty
        while (head->next != NULL)
            head = head->next;
    }
    return head;
}
```



Figures for linked list are from http://www.zentut.com/c-tutorial/c-linked-list/

Append

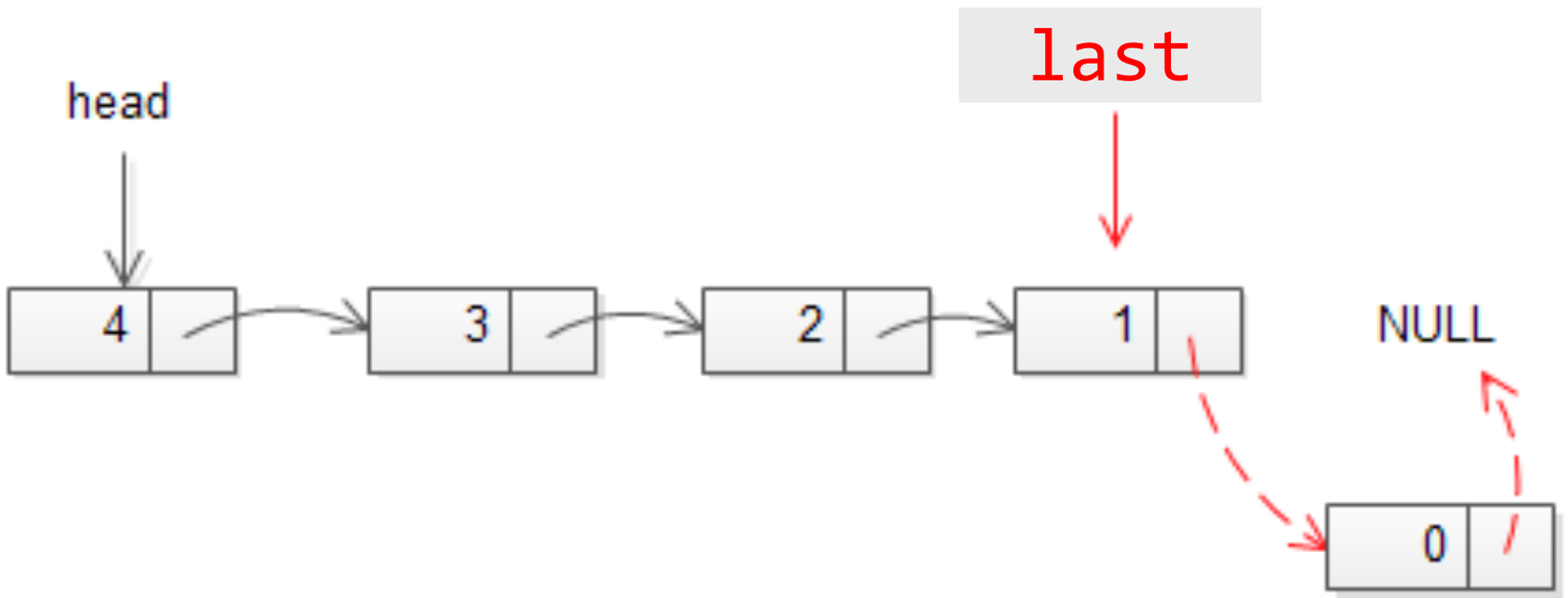
```
node * append(node * head, node * newnode)
{
    // How?
}
```



Figures for linked list are from http://www.zentut.com/c-tutorial/c-linked-list/

Append

```
node * append(node * head, node * newnode) {
    node *last = find_last(head);  // find the last one
    if (last == NULL)    // if the list is empty, newnode is the head
        return newnode;
    last->next = newnode;
    newnode->next = NULL;
    return head;      // return the (unchanged) head
}
```



Figures for linked list are from http://www.zentut.com/c-tutorial/c-linked-list/

Enumeration types (ABC 7.5)

- User-defined integer-like types:

```
typedef enum {
  Red, Orange, Yellow, Green, Blue, Violet
} Color;
```

- Names look like C identifiers
  - are listed (enumerated) in definition
  - treated as integers
    - can add, subtract, compare (`Red + Green, if(color==Red)`…)
    - can't print as symbol
    - but debugger generally will

Enumeration types


```
// enum start from 0 by default
enum week {Sun, Mon, Tue, Wed, Thur, Fri, Sat};
enum week dow = Mon;
```

```
// But can be initialized; Warning is 2, Error is 3, etc.
enum status {OK = 1, Warning, Error, Fatal};
```

Type qualifier: const

```
// constant int
const int   a = 10;       // cannot change a
// a pointer to a constant int
const int  *pa = &a;  // can change pa, but not *pa
// a constant pointer to an int
int  * const pb = &b;     // can change *pb, but not pb
// a constant pointer to a constant int
const int  * const pc = &a; // cannot change *pc or pc

// cannot change the source string
char * strcpy(char * dest, const char * src);
```

Function pointers


```
/* function returning integer */
int func();

/* function returning pointer to integer */
int * func();

/* pointer to function returning integer */
int (*func)();

/* pointer to function returning pointer to int */
int * (*func)();
```

Pointer to function example

```
int     mymax(int a, int  b)
{
    return (a > b) ? a : b;
}
// a pointer to function
int (*pf)(int a, int b);

// assign a value to the pointer
pf = mymax;          // C99 style. Note that it is NOT mymax()
pf(3, 5);
pf = & mymax;
(*pf)(3,5);
```

Use of function pointers

- Call-back mechanism
  - Generic functions (example coming next)
  - pthread_create()
  - Dynamic signal handlers,…

- You can store function pointers in arrays
  - And arrays stored in structures!
  - And you can simulate objects in Object Oriented Languages!

Example: quicksort in C library

- The prototype (in <stdlib.h>)

```
void qsort(void * base,
           size_t nel,
           size_t width,
           int (*compare)(const void *, const void *));
```

- qsort takes…
  - base: the address of the array as an untyped pointer
  - nel: the number of elements in the array
  - width: the size (in byte) of ONE element of the array
  - compare: a **pointer to a function** that compares two values

qsort() only knows it is asked to sort nel items, each having width bytes.
It does not know the type of elements or how to compare them.

Why passing a function to qsort?

- Need to tell qsort() how to compare items in the array
  - We have a *generic* quickSort implementation
    - Do not want to implement one for each type of data
- The qsort() implementation calls the comparator to rank elements

```
int (*compare)(const void *a, const void *b);
```

- The function takes the address of two items to be compared,
- and returns:
  - 0 if *a EQUALS *b
  - A positive value if *a is GREATER THAN *b
  - A negative value if *a is LESS THAN *b

Example of compare() function

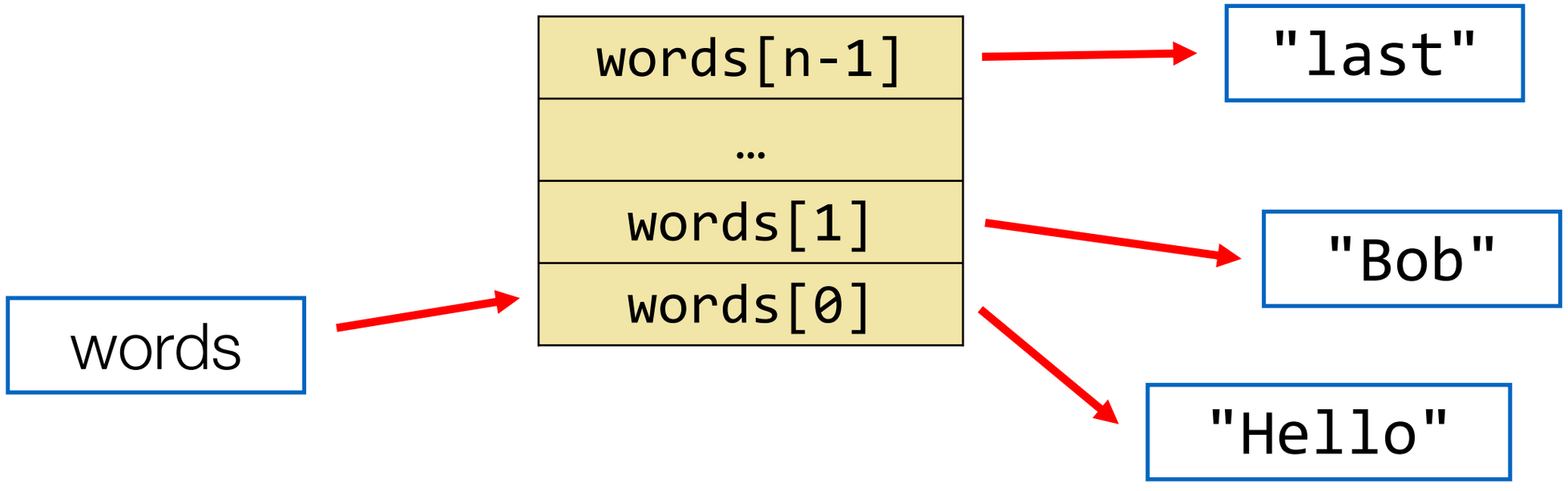
When qsort() needs to compares two items, it provides their addresses

```
int compare_int(const void *a, const void *b)
{ // qsort() does not know the type, but you know
   return *(int *)a - *(int *)b;
}
int compare_double(const void *a, const void *b)
{ // qsort() does not the type, but you know
   double va, vb;
   va = *(double *)a; vb = *(double *)b;
   return va > vb ? 1 : (va < vb ? -1 : 0);
}
```

Example: sort array of strings

- Element are pointers to strings
  - Need to compare string, instead of pointer

```
int compare_string(const void *a, const void *b) {
// how to compare *a and *b ?
// for example, a is &words[0] and b is &word[1]
```



Compare string pointers

An element in array words is (char *).

a is the address of an element of type (char *). So, a's type is (char *) *

```
int compare_string(const void * a,const void * b)
{
   char *s1, *s2;
   s1 = *(char **)a;     s2 = *(char **)b;

   return strcmp(s1, s2);  // use library function to compare
}

// or on one line
int compare_string(const void * a,const void * b)
{
   return strcmp(*(char**)a,*(char**)b);
}
```

Calling quicksort()

```
int compare_string(const void* a,const void* b)
{
    return strcmp(*(char**)a,*(char**)b);
}


int some_function(void)
{
    …
    char** words = malloc(sizeof(char*)*n);
    …
    qsort(words,n,sizeof(char*),compare_string);
    …
}
```

Type casting to char ** before dereferencing