Computer Architecture

Paul-André Melliès

Lecture 7 : Programming in C

D. explicit allocation and linked lists

```
struct matrix *allocate_matrix (int width, int height){
  struct matrix *pm;
  int i;

  // (1) allocate the structure
  pm = malloc (sizeof(struct matrix));
  pm -> width = width;
  pm -> height = height;

  // (2) allocate the array of the line addresses
  pm -> content = malloc (height * sizeof(int *));

  // (3) allocate the lines
  for (i = 0; i < height; i++);
    pm -> content [i] = malloc(width * sizeof(int));

return pm; // return the address of the allocated matrix
}
```

Explicit allocation

Explicit allocation of memory

```
#include <stdio.h>   // necessary to use printf, sizeof
#include <stdlib.h>  // necessary to use malloc, free

int main(){
  int *p;

  // allocate a region of ten times the size of int
  // store the address of the allocated region in p
  p = malloc(10 * sizeof(int));

  p[0] = 38;
  p[1] = 42;
  p[2] = p[1] + 3;

  // ... sequence of instructions ...

  // free the region of allocated memory
  free(p);
  return 0;
}
```

The `malloc` function

```
malloc(10 * sizeof(int))
```

- calls the function `malloc` defined in the library `stdlib`
- sends as argument the size of memory to allocate

An equivalent instruction : `malloc(sizeof(int[10]))`

The function `malloc` does two things :

- it allocates a region in memory of the required size (in a single block, and not initialized)
- it returns as a pointer the address of the first byte of the allocated region.

Typically, the instruction

```
p = malloc(10 * sizeof(int));
```

stores the address of the allocated memory in the pointer `p` .

Memory leak

`free(p)`

deallocates the memory region allocated by `malloc`

A typical error to avoid : « memory leak »

```
p = malloc(10 * sizeof(int));
// ... no free between two malloc
p = malloc(10 * sizeof(int));
```

The address of the region allocated by the first `malloc` has been lost...
It will be impossible to free this region before the end of the execution !!!

Too much memory allocated and not freed ⟶ not enough memory

Failure of allocation

Allocation can fail when not enough memory is available.
In that case, the function `malloc` returns `NULL` (the « null » pointer)

In principle, one should always treat the case of failure of allocation :

```
p = malloc(10 * sizeof(int));
if (p == NULL) {
  printf ("function ... : error of allocation.\n");
  exit(EXIT_FAILURE);
}
```

`exit (EXIT_FAILURE);` interrupts the program
with the predefined value `EXIT_FAILURE`
transmitted to the shell (which will then handle the error)

Variants of `malloc`

```
p = calloc(10,sizeof(int));
```

Equivalent to `p = malloc(10*sizeof(int));`
except that the allocation memory is initialized with null values.

```
p = malloc(10 * sizeof(int));
p = realloc(p, 20 * sizeof(int));
```

- modifies the size of the allocated region of memory
  (keep the data for a larger size, truncate them for a smaller size)
- the reallocation may occur at a different address
  (in that case, the old region is freed, the value of `p` changes
  and the content of the old memory is copied to the new one)

Return type of `malloc` and generic pointers

The address returned by `malloc` may be assigned to any pointer type

```
char *p;      p = malloc(10 * sizeof(char));
int *q;       q = malloc(10 * sizeof(int));
int **r;      r = malloc(10 * sizeof(int *));
```

The return type of `malloc` is `void *` (or « pointer to `void` »)

`void` : the « empty type » is a type without any value

`void *` : type of pointers without any specification about the size of objects : « generic pointers »

The casts (pointer ⟷ generic pointer) are implicit but they can be made explicit :

```
int *p;       p = (int *)malloc(10 * sizeof(int));
```

Applications

Exercise 1

A question apparently imprecise, at least at first sight :

Write a function which takes as argument a character string `s`
and returns either :

- the string consisting of the first word of `s` when `s` is not empty
- the empty string otherwise

The meaning of « a function which takes as argument a string » is clear :
the function should take as argument a pointer `s` of type `char *`
which contains the address of a character string -- that is :
an array containing a sequence of characters ended by `'\0'`

But how should we understand « return a character string » ?

Exercise 1

How should we understand « return a character string » ?

- Return the « value » of an array containing the string ?
  Simply impossible !

- Store the string in a global array in which the caller will read the result ?
  Well, can be done... but should be avoided if possible.

```
int m[256]; // 256, that should be enough
void first_word (char *s) {
// ... store the first word in m
}
```

The point is that the current content of `m` depends on the sequence of function calls, so if several functions can store characters in `m` the program may become difficult to read and debug.
Moreover, the size of `m` may be inadapted.

Exercise 1

How should we understand « return a character string » ?

- Pass to the function the address of an array in which the function will store the character string ?

This can be done of course... but :

```
void first_word (char *s, char *m) {
// ... store the first word in m
}
```

Not so different from the previous code, except for the locality of `m` and the size of the array `m` may not be sufficient.

Exercise 1

How should we understand « return a character string » ?

☞ Return the address of an array in which the function store the character string ?
Yes !

But the address of what array ?
Certainly not the address of a local array... why ?

```
char *first_word (char *s) {
char m[256];  // 256 should be sufficient...
// ... store the first word in m
return m;  // error !!!
}
```

Exercise 1

How should we understand « return a character string » ?

Return the address of an array allocated by `malloc`
and containing the string consisting of the first word.

Yes !

- a (partial) exploration of the input string will be enough to know the size of the word and to allocate the right amount the appropriate amount of memory
- in contrast to a local array, the array will persist in memory when the function returns
- caller will thus be able to read the result and free the array by `free` as soon as the data is not needed anymore.

Solution

```
char *first_word(char *s){
  char *m;
  int start, length, i;
  // start and length of the first word

  // compute the position of start
  for (start = 0; s[start] == ' '; start++);

  // compute the length of the first word
  for (length = 0; s[start+length] != ' ' && s[start+length] != '\0'; length++);

  m = malloc ((length + 1) * sizeof(char));  // +1 for '\0'

  for (i=0; i<length; i++)                   // copy of the first word
    m[i] = s[start + i];

  m[length] = '\0';                          // store the null character
                                             // at the end of the string
  return m;
}
```

Exercise 2 Java-like matrices (more difficult)

```
struct matrix {
   int width;
   int height;
   int **content;  // array of the addresses
};
```

The field `content` is the address of an array of addresses
(an array of pointers to `int`)

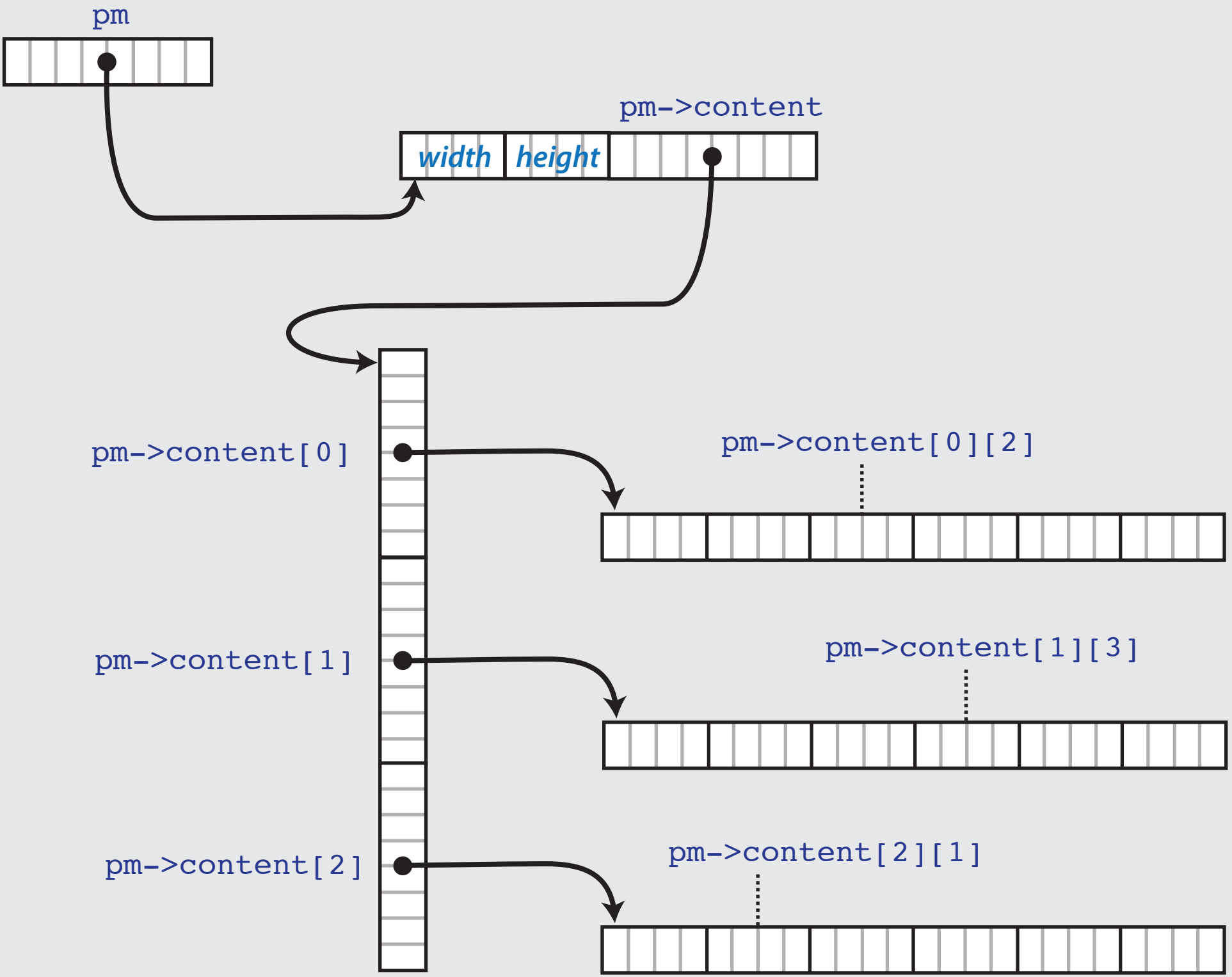
The type of the field is « pointer to pointer to `int` » or : `int **`

Each element of this field is the address of an array containing one of the lines of the matrix.

Exercise 2 Java-like matrices (more difficult)

When `pm` is a pointer of type `matrix` :

`pm -> content`	address of the array of the lines
`pm -> content[i]`	address of the line `i` of type `int *`
`pm -> content[i][j]`	content of position `(i,j)` of type `int`


pm
pm->content
width
height
pm->content[0]
pm->content[0][2]
pm->content[1]
pm->content[1][3]
pm->content[2]
pm->content[2][1]

Exercise 2 Java-like matrices (more difficult)

Function allocating a matrix in memory :

```
struct matrix *allocate_matrix (int width, int height){
  struct matrix *pm;
  int i;

  // (1) allocate the structure
  pm = malloc (sizeof(struct matrix));
  pm -> width = width;
  pm -> height = height;

  // (2) allocate the array of the line addresses
  pm -> content = malloc (height * sizeof(int *));

  // (3) allocate the lines
  for (i = 0; i < height; i++);
    pm -> content [i] = malloc(width * sizeof(int));

return pm; // return the address of the allocated matrix
}
```

Exercise 2 Java-like matrices (more difficult)

Function cloning a matrix :

```
struct matrix *clone_matrix (struct matrix *pm){
  struct matrix *pmc;
  int i, j;

  // (1) allocate the structure of the clone
  // and of its array of line addresses
  pmc = malloc (sizeof(struct matrix));
  pmc -> width = pm -> width;
  pmc -> height = pm -> height;
  pmc -> content = malloc(pm -> height * sizeof (int *));

  // (2) allocate the lines, copy the lines of the original
  for (i = 0; i < pm -> height; i++){
    pmc -> content[i] = malloc(pm -> width * sizeof(int));
    for (j = 0; j < pm -> width; j++){
      pmc -> content[i][j] = pm -> content[i][j];
    }
  }
return pmc; // return the address of the clone matrix
}
```

Linked lists

Lists in C

A list is a finite sequence of elements of the same type :

[3 , 17 , 38 , 3 , 12]

Adding an element in the list is performed « on the left »

3 :: [17 , 38 , 3 , 12] ⟶ [3 , 17 , 38 , 3 , 12]

The « head » of a list is its first element

The « tail » of a non-empty list is what follows the head :

[3 , 17 , 38 , 3 , 12] ⟶ 3 :: [17 , 38 , 3 , 12]

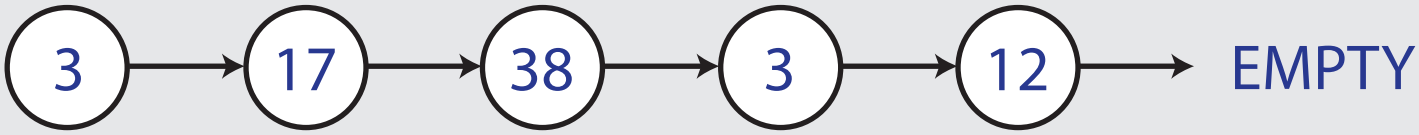
head tail

Lists in C

A list may be also seen as a graph
obtained by plugging together the elements of the list :

[3 , 17 , 38 , 3 , 12]

3 :: (17 :: (38 :: (3 :: (12 :: [ ]))))

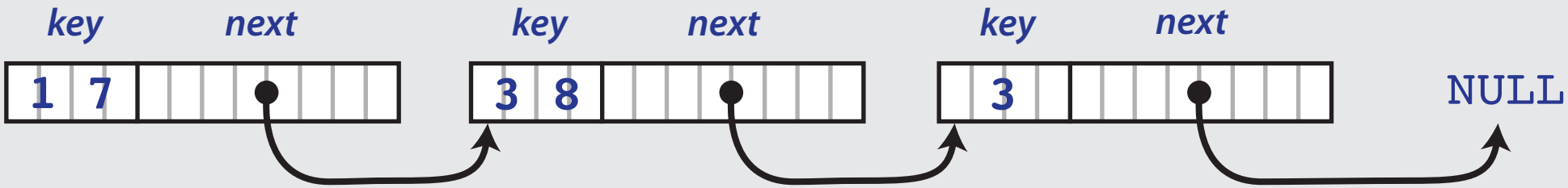


a node of the graph is called a « cell »
its label is called the « key » of the node

Linked lists

The cells are represented by *structures* containing the key as well as a pointer to the following cell.

Typically, the list `[ 17 , 38 , 3 ]` is represented in memory as :



Structure type to represent the cells :

```
struct cell = {
  int key;
  struct cell * next;
};
```

Linked lists

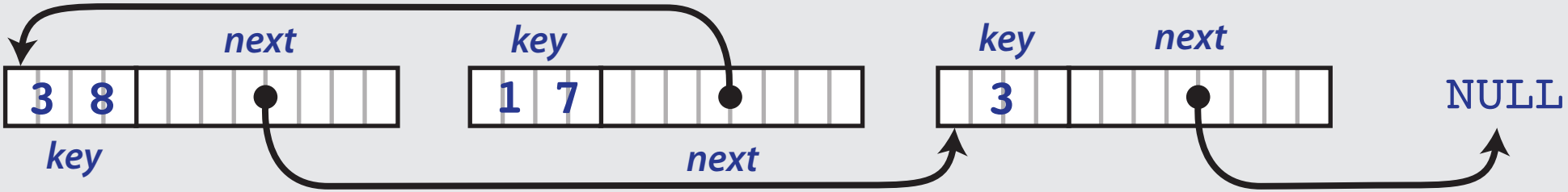
Its last cell has its pointer `next` equal to `NULL`

A list will be represented in the type `(struct cell *)` by :

- the `NULL` pointer if the list is empty
- a pointer to its first cell if the list is non empty

For that reason, the field `next` always represents a list :
in particular, the field `next` of the last cell represents the empty list.

Note that the cells are connected together but they may appear anywhere in memory (a list is not an array !)



A useful notation

Remember that :

`(* pointer).field` can be also written `pointer->field`

So :

`cellptr -> key` means `(*cellptr).key`

`cellptr -> next` means `(*cellptr).next`

`cellptr -> next -> next` means `(*((*cellptr).next)).next`

etc ...

Exploration of a linked list

At the beginning of the program :

```
struct cell {
  int cell;
  struct cell *next;
};
```

`struct cell *cellptr = ....;` with `cellptr` (pointer to cell)

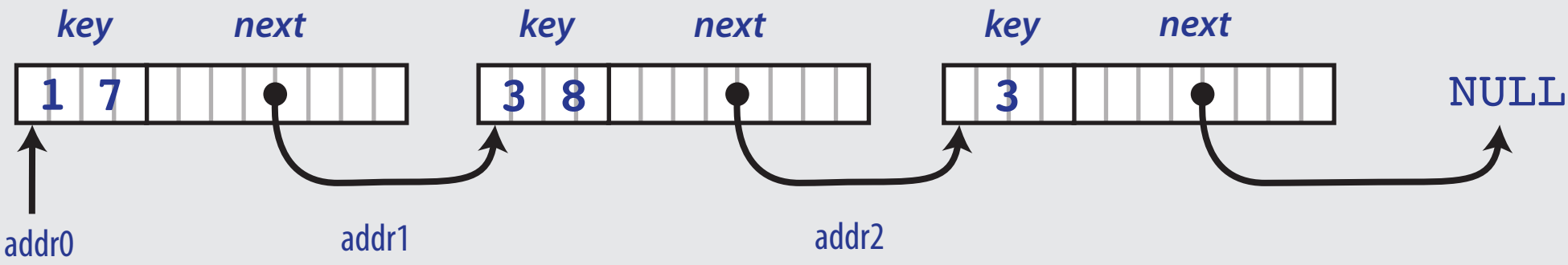
- equal to `NULL` or
- equal to the address of the first cell of a (non empty) list.

If `cellptr` is not equal to `NULL` then :

`*cellptr`	is of type	`(struct cell)`	(pointed cell)
`cellptr->key`	is of type	`int`	(key of the cell)
`pc->next`	is of type	`(struct cell *)`	(pointer to the next cell)

Illustration

```
struct cell *cellptr = ...;       // cellptr initialized at addr0
printf("%d ", cellptr -> key);    // print 17
cellptr = cellptr -> next;        // cellptr is equal to addr1
printf("%d ", cellptr -> key);    // print 38
cellptr = cellptr -> next;        // cellptr is equal to addr2 etc.
......
```



Illustration

```
void print_list (struct cell *cellptr){
  if (cellptr = NULL) {
    printf("the list is empty\n");
    return;
  }

  // as long as the empty list is not reached
  while (cellptr != NULL){
    printf("%d", cellptr->key);  // print the key
    cellptr = cellptr -> next;   // move to next cell
  }
  printf("\n");    // go to the next line
}
```

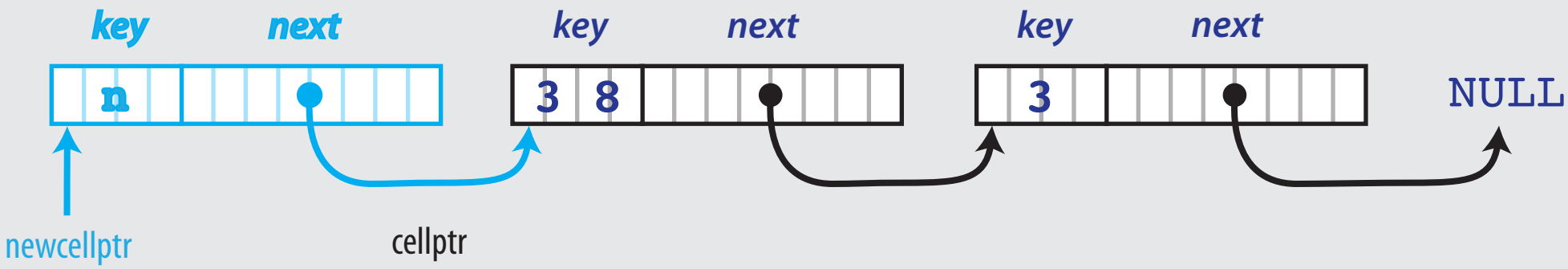
Illustration

```
int length_list (struct cell *cellptr){
  int length = 0;

  // as long as the empty list is not reached
  while (cellptr != NULL){
    length++                        // increment length
    cellptr = cellptr -> next // move to the next cell
  }

  return length;
}
```

Append an element to a list



Adding a new element `n` to a linked list is done in three steps :

- use `malloc` to allocate a new region of memory :

```
struct cell *newcellptr;
newcellptr = malloc(sizeof(struct cell));
```

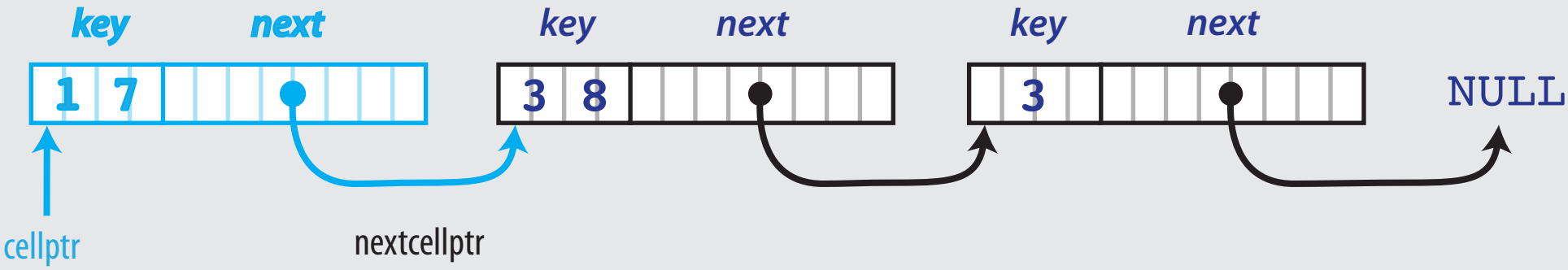
- give the value `n` to `newcellptr->key`
- give the value `cellptr` to `newcellptr->next`

Append an element to a list

```
struct cell *append_list(struct cell *cellptr, int n){
  struct cell *newcellptr;
  newcellptr = malloc(sizeof(struct cell));
  newcellptr -> key = n;
  newcellptr -> next = cellptr;
  return newcellptr;
}

int main(){
  struct cell *cellptr = NULL;          // empty list [ ]
  cellptr = append_list(cellptr, 3)     // [3]
  cellptr = append_list(cellptr, 17)    // [17, 3]
  cellptr = append_list(cellptr, 38)    // [38, 17, 3]
  ...
return 0;
}
```

Remove the first element of a list



Removing the first element from a non empty list is done in two steps :

- remember the address of the tail of the list :

```
struct cell *nextcellptr;
nextcellptr = cellptr -> next;
```

- use `free` to deallocate the first cell of the linked list :

```
free(cellptr);
```

Remove the first element from a list

```
struct cell *tail_list(struct cell *cellptr){
  struct cell *nextcellptr;

  if (cellptr == NULL){
    printf("The list is empty\n");
    exit(EXIT_FAILURE);
  }
  nextcellptr = cellptr -> next;
  free(cellptr);
  return nextcellptr;
}

int main(){
  struct cell *cellptr = NULL;          // empty list [ ]
  cellptr = append_list(cellptr, 3)     // [3]
  cellptr = tail_list(cellptr)          // empty list [ ]
  ...
return 0;
}
```

Remark

A similar treatment of binary trees of integers is possible :

```
struct node {
int key;
struct node *left;
struct node *right;
};
```

A tree is a pointer to type `node`

- the `NULL` pointer when the tree is empty
- a pointer to the root node of the tree when the tree is nonempty

The functions on the lists are naturally defined by recursion :
recursion becomes somewhat necessary to manipulate trees.

Acknowledgements and references

The slides of this lecture are largely based on a very nice and cleverly crafted introductory course on the programming language C given by Vincent Padovani in my University Paris Diderot.

Padovani is a pedagogical master and I am greatly endebted to his art here !

I have also inserted in the lecture a series of slightly more demanding examples extracted from the marvelous and so instructive book by Kernighan and Ritchie.

Please read these programs carefully and try to understand what they do...
You will discover their beauty, and learn a lot from them !