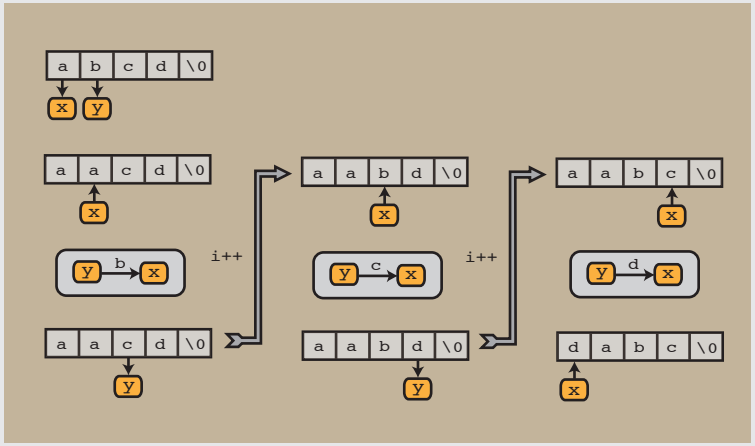
Computer Architecture

Paul-André Melliès

Lecture 5 : Programming in C

B. arrays, strings and functions



Arrays

Declaring an array

```
int t[4], i ;

// write a value in each element of the array
t[0] = 12;
t[1] = 2;
t[2] = 4;
t[3] = 7;

// read and print the content of each element
for (i=0; i<4; i++)
    printf("content of element no %d : %d\n",i,t[i]);
```

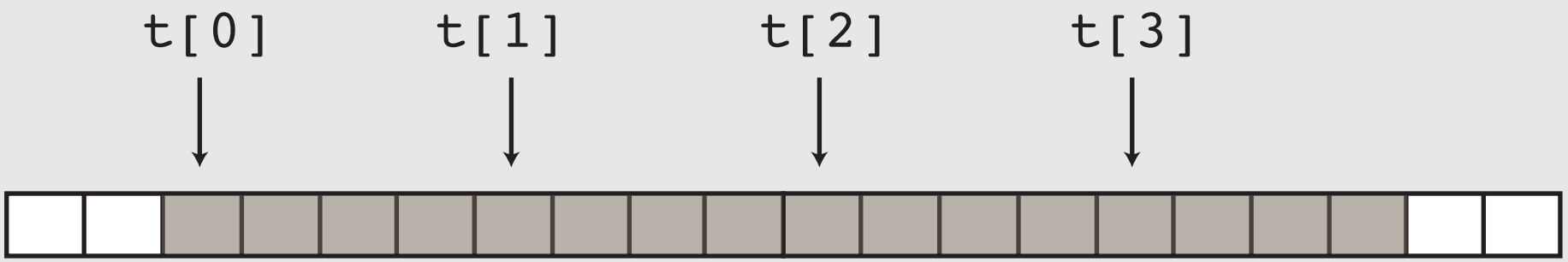
☞ `int t[4];`

allocates a piece of memory sufficiently large to contain four integers :

4 x 4 bytes = 16 bytes = 4 x 32 bits = 128 bits

the variable `t` is of type : array of integers containing 4 elements

The four elements of the array of integers are accessed in memory in the following way :



Each `t[i]` for `i` = 0,1,2,3 is then treated as a variable of type integer.

Important : the first element of the array is indexed by 0 rather than 1.

Arrays

An element of the array `t` may be also indexed by an expression of type integer :

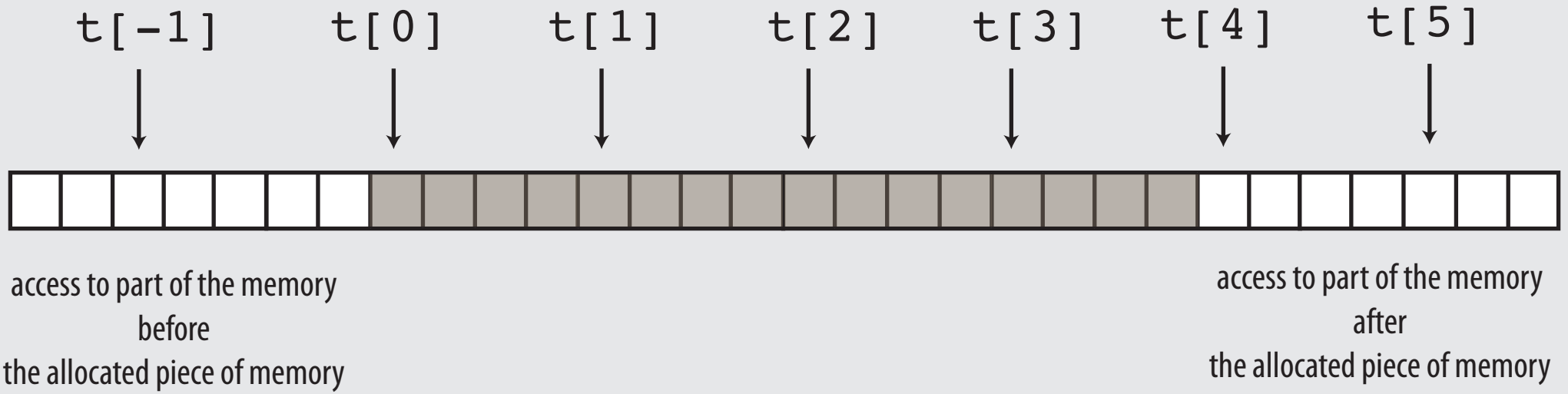
```
t[integer expression]
```

Basic illustration :

```
int t[4], i ;

// write a value in each element of the array
t[0] = 12;
t[1] = 2;
t[2] = 4;
t[3] = 7;

// read and print the content of each element
for (i=0; i<4; i++)
    printf("content of element no %d : %d\n",i,t[i]);
```

Arrays

Warning : no verification is made that you are accessing an element allocated in memory during the declaration of the array :



This may lead the program to

- read or write ***another variable*** of the program
- or have access to some ***protected part*** of the memory.

In that latter case, the program crashes with a signal from the OS :

```
Segmentation fault (core dumped)
```

Immediate initialization of an array

- `int t[4];`

The values of the four integer variables are not determined

- `int t[] = {12, 2, 4, 7};`

The number `4` of elements of the array is deduced

- `int t[4] = {12, 2};`

The array contains 4 elements, with

- `t[0]` initialized to 12
- `t[1]` initialized to 2
- `t[2]` initialized to 0
- `t[3]` initialized to 0

- `int t[100] = {};`

The array contains 100 elements, all initialized to 0.

Arrays : an usual kind of « variable » ...

- It is not possible to reassign the value of an array :

```
t = ... // cannot be compiled
```

- The expression `(t == u)` has a meaning (discussed later) but beware : it does not compare the content of the arrays.

  So, in order to compare the content of two arrays `t` and `u` one needs to compare them element by element.

- It is not possible to alter the size of an array after declaring it. since this would mean altering its type :

an array of integers of size 4	`int[4]`
is not the same thing as	
an array of integers of size 5	`int[5]`

Higher-dimensional arrays

Higher-dimensional arrays

```
int m[3][4];
```

This declares an array of arrays of integers of 4 elements.

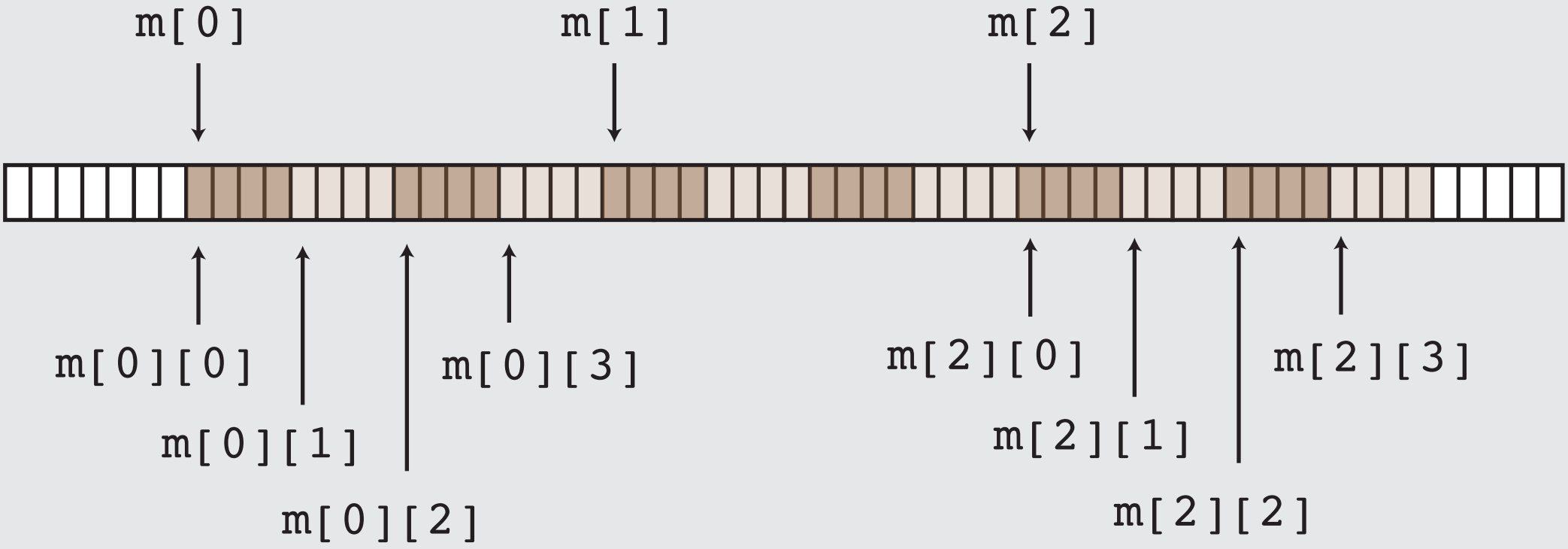
More precisely : an array of three elements where
each element is itself an array of integers of four elements :

- `m[0]` is a first array of four elements
- `m[1]` is a second array of four elements
- `m[2]` is a third array of four elements

The declaration allocates a piece of memory containing

3 x 4 x 4 bytes = 48 bytes = 12 x 32 bits

Higher-dimensional arrays : organisation of the memory



Warning : again, no verification of the indices...

Higher-dimensional arrays : immediate initialization

```
int m[3][4] =
    {{1,0,0,0},     // content of m[0]
     {0,1,0,0},     // content of m[1]
     {0,0,1,0}};    // content of m[2]
```

As before, the initialization of arrays may be partial and all unspecified elements are initialized to 0 :

```
int m[3][4] =
    {{1,0},         // m[0][2] and m[0][3]
                    // initialized to 0
     {0,1,0,0}};    // content of m[1] specified
                    // elements of m[2]
                    // all initialized to 0
```

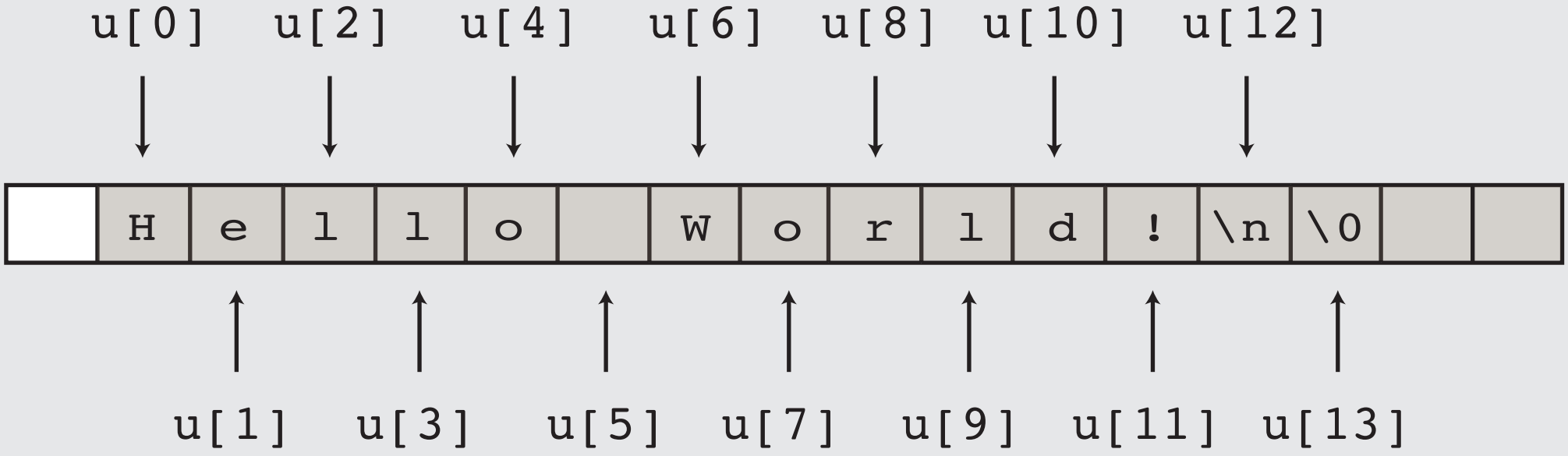
Character arrays

Character arrays and strings

```
char u[100];  // array of char sufficiently large
              // could be 256 or 1000, etc...

// store the string "Hello World!\n" in the array

u[0] = 'H';
u[1] = 'e';
u[2] = u[3] = u[9] = 'l';
u[4] = u[7] = 'o';
u[5] = ' ';
u[6] = 'W';
u[8] = 'r';
u[10] = 'd';
u[11] = 33 ;
u[12] = '\n';
u[13] = '\0';

printf("print the char array : %s",u);
// print the string : Hello World!
```

Character arrays and strings

☞ Location of the character array `u` in memory :



Remark :
this should be compared with the representation of an array of integers.

Character arrays and strings

☞ A string of characters is represented as a character array.

The end of the string is indicated by a character `\0` in the array.

The character `\0` is called the « null character » and its ASCII value is 0.

Quite obviouly, the null character is not printed on the screen.

☞ `printf("%s", u)`

Inserts at the `%s` all the characters in the array between `u[0]` and the first encountered element with value `\0` in the array.

The result may be quite surprising when the array `u` is not initialized !!!

Character arrays and strings

Typical shape of a program processing a string of characters :

```
for (i=0; u[i] != '\0'; i++) {

     // any treatment of the string
     // character after character
     // for instance, print :

     printf("%c", u[i]);
}
```

Immediate initialization of a character array by a string of characters

☞ `char u[100] = "abc";`

The character array `u` is allocated with a size of 100 elements

`u[0]` is initialized to `'a'`
`u[1]` is initialized to `'b'`
`u[2]` is initialized to `'c'`

All the other elements of the array are initialized to the value `'\0'`

In particular, the element `u[3]` is initialized to the value `'\0'` and thus indicates the end of the string of characters.

Immediate initialization of a character array by a string of characters

☞ `char u[] = "abc";`

The character array `u` is allocated just with the appropriate size... which means in that case with a size of 4 = 3+1 elements.

`u[0]` is initialized to `'a'`
`u[1]` is initialized to `'b'`
`u[2]` is initialized to `'c'`

`u[3]` is initialized to `'\0'` to indicate the end of the string.

Read a string from the keyboard

```
char u[100], c;

printf("enter a string of characters : ");

// wait for a string and store it in the array u
scanf("%s",u);
printf("String just read : %s\n", u);

// wait for a character and store it in the char c
scanf(" %c",&c);
printf("Character just read : %c\n", c);
```

Note that there is no & in the instruction :

```
scanf("%s",u);
```

The reason is that `u` already denotes the « address » of the array.

Read a string from the keyboard

```
char u[100];

printf("Enter a string of characters : ");

// wait for a string and store it in the array u
scanf("%s",u);

printf("String just read : %s\n", u);
```

```
Enter a string of characters : Marvelous afternoon
String just read : Marvelous
```

The `scanf` instruction :

- starts from the first character different from the space character
- stops at the first encountered space (and at the end of the string otherwise)
- adds a null character at the end of the string
- and throws **an abort exception** when the array is not sufficiently large...

Exercise

Write a program which asks the user to enter a string and then computes and prints the length of the string.

Solution

```
#include <stdio.h>

int main(){

char u[1000];
int i;

// ask user to enter a string
printf("Enter a string of characters : ");
scanf("%s", u);

// start from the first element of the array u
// and increment i for each element of the array
// until the for loop encounter the null character
for (i=0; u[i] != '\0'; i++);

// print the number of characters in the string
printf("Number of characters : %d\n", i);

return 0;
}
```

Exercise 2

Write a program which asks the user a string as well as a character and then computes and prints the number of times this character occurs in the string.

Solution

```
#include <stdio.h>

int main(){
char c, u[1000];
int i,counter=0;

// ask user to enter a string
printf("Enter a string of characters : ");
scanf("%s", u);

// ask user to enter a character
printf("Enter a character : ");
scanf(" %c", &c);

for (i=0; u[i] != '\0'; i++){ // explore the elements of u
    if (u[i] == c){           // count the number of c's
    counter++;
    }
}

// print the number of characters in the string
printf("Number of characters %c : %d\n", c, number);
}
```

Alternative solution

The for loop

```
for (i=0; u[i] != '\0'; i++)
{
    if (u[i] == c){
    counter++;
    }
}
```

may be rewritten more concisely as :

```
for (i=0; u[i] != '\0'; i++){counter += (u[i]==c);}
```

Can you explain why ?

Exercise 3

Write a program which asks the user to enter a string of characters and then writes the string in the reverse order !

Note :
the purpose of the program is not to reverse the string of characters but simply to **write it** in the reverse order.

Analysis of the exercise

Since the task of the program is only to print in reverse order, we can simply :

1. go to the end of the string and then
2. read the string backwards and print each character encountered.

What we will need :

- the character array itself `u`
- a loop with counter `i` to count the number of characters until one reaches the first null character in the array `u`
- another loop to come back to the beginning of the string.

Solution

```
#include <stdio.h>

int main(){
char u[1000];
int i;

// use scanf to get the string from user
...

for (i=0; u[i] != '\0'; i++);
// i is equal to the first position of the null character

// the initialization of the for loop sets the integer i
// at the position of the last character of the string
// one gets i = -1 when the string is empty
for (i = i-1; i >= 0; i--){
    printf("%c", u[i]);
    }

// print the number of characters in the string
printf("\n");
}
```

Another possible solution

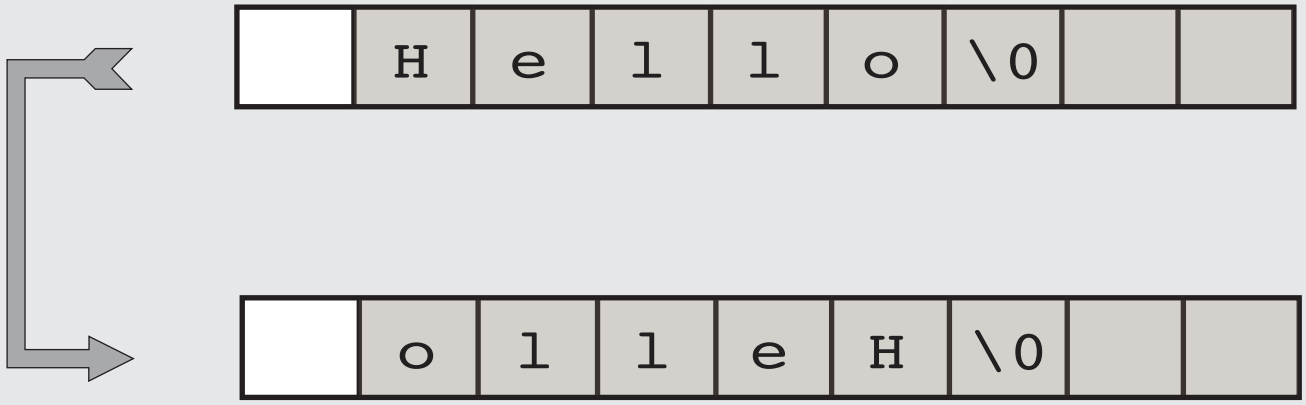
(slightly more advanced, to be explained later)

```
#include <stdio.h>

int printreverse(char u[1000], int i)
{
  if (u[i]!='\0')
    {
    printreverse(u,i+1);
    printf("%c",u[i]);
    };
  return 0;
}

int main(){
  char u[1000];

  printf("enter your string : ");
  scanf("%s",u);
  printreverse(u,0);
  printf("\n");
  return 0;
}
```

Exercise 4

Write a program which asks the user to enter a string of characters then reverses the string and prints it !

More difficult, this time, the purpose of the program is to reverse the string itself :



Solution

```
#include <stdio.h>

int main(){
int i, length;
char u[1000], tmp;

// use scanf to get the string from user
...

for (length=0; u[length] != '\0'; length++);
// length is equal to the length of the string

for (i = 0; i < length/2; i++){
    tmp = u[i];
    u[i] = u[length - 1 - i];
    u[length - 1 - i] = tmp;
}

printf("%s", u);
}
```

Exercise 5 left rotation of a string

The left rotation of a non-empty string is the string obtained by shifting to its first character to the end of the string.

`abcde`		`bcdea`
`ab`		`ba`
`a`		`a`

Write a program which :

- reads a string of character
- stores it in a character array `u`
- and if the string is not empty... transforms it (inside the same array) into its left rotation
- prints the resulting string in the end.

Solution

```
#include <stdio.h>

int main(){
  int i;
  char u[1000], c;

  // use scanf to get the string from user
  ...

  if (u[0] != '\0')
    {
      c = u[0];   // store the first character of the string

      for (i = 1; u[i] != '\0'; i++)
          u[i - 1] = u[i];

      // at this stage the counter i describes the position
      // of the first null character of the array
      // replace the last character of the string by c
      u[i - 1] = c;
    }

  printf("The reserved string is : %s\n", u);
}
```

Exercise 6

More difficult :

Write a program which asks user to enter a text
and then computes (and prints) the number of words in the string.

Conventions :

- the words are separated by spaces
- there can be several spaces between two words
- there can be several spaces at the beginning of the string
- the string can be empty or contain only spaces

Additional rule :

- the string can be read only once

Analysis of the exercise

Every position `i` in the string `u` can be :

- either on a word
- or on a space

In the array `u` an index `i` contains **the last letter of a word** precisely when :

- `u[i]` does not contain a space character (hence it is on a word)
- `u[i+1]` contains :
  - either a space character `' '`
  - or the null character `'\0'`

It is thus sufficient to count the last letters of a word.

Remark :

It would be also possible to count the first letters of a word.
How would you proceed ?

Solution

```
#include <stdio.h>

int main(){
  int i, numberofword=0;
  char u[100];

  // wait for a string and store it in the array u
  // use scanset character [^\n] to read input with spaces
  scanf("%[^\n]s",u);
  printf("String just read : %s\n", u);

  // compute the number of words
  for (i = 0; u[i] != '\0'; i++){
      if ( u[i] != ' ' &&                          // not a space
          (u[i+1] == ' ' || u[i+1] == '\0'))       // space or null
          numberofword++;                          // add a word
  }

  printf("The number of words is : %s\n", numberofword);
}
```

Exercise 7 right rotation of a string

The right rotation of a non-empty string is the string obtained by shifting to its first character to the end of the string.

`abcde`		`eabcd`
`ab`		`ba`
`a`		`a`

Write a program which :

- reads a string of character
- stores it in a character array `u`
- and if the string is not empty... transforms it (inside the same array) into its right rotation
- prints the resulting string in the end.

Analysis of the exercise

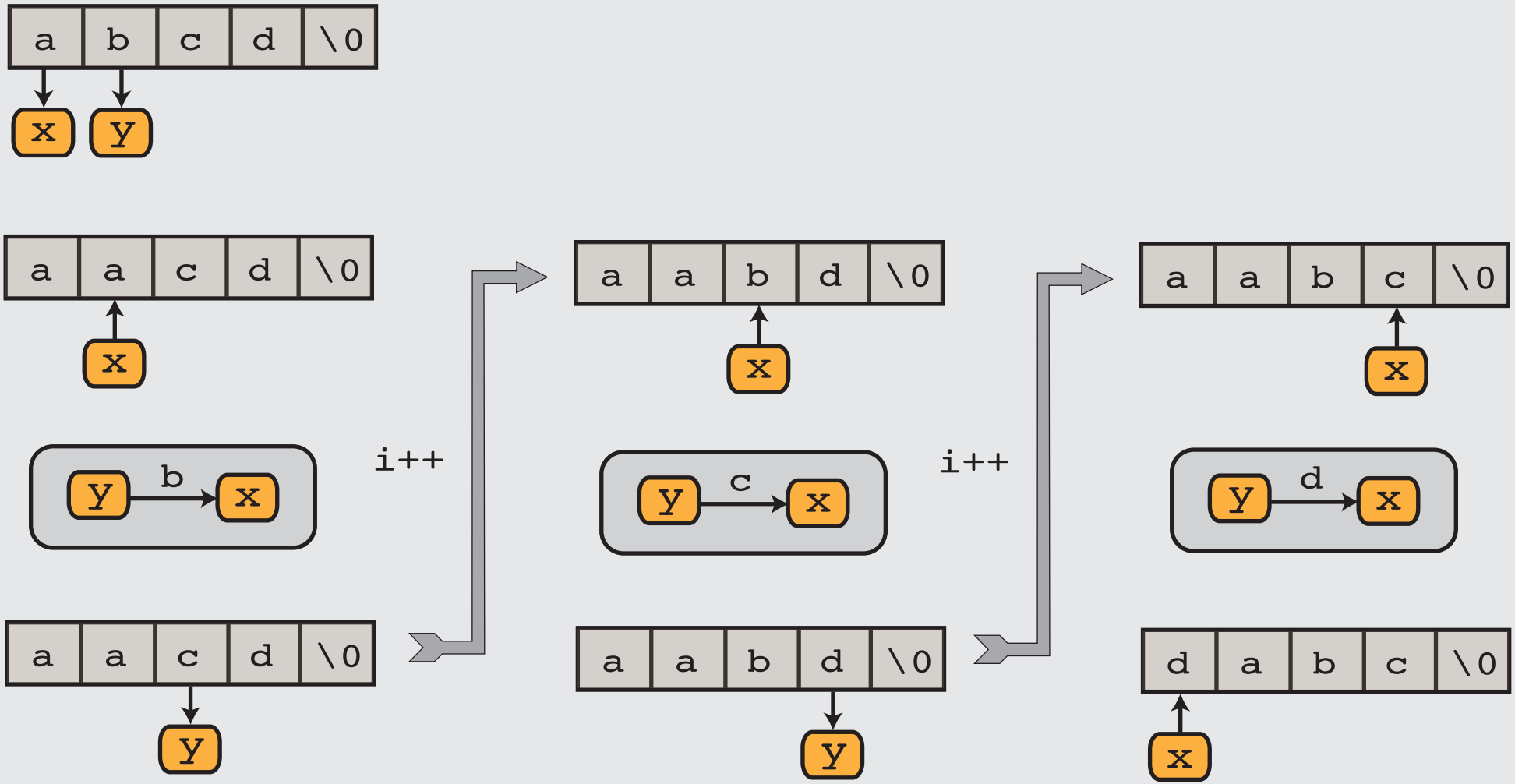
Every character in the string should be shifted to the right...

Beware :
if you are not careful, each character will overwrite the next one
and the second character `b` will be lost !

How much does one need to remember during the loop ?
And on which order should one proceed ?
This is much more subtle than for the left rotation !

Indication :
One needs two variables for keeping track of the relevant information.

Analysis of the exercise



Solution

```
#include <stdio.h>

int main(){
  int i;
  char u[1000], x, y;

  if (u[0] != '\0'){
      x = u[0];
      for (i = 1; u[i] != '\0'){
          y = u[i];
          u[i] = x;
          x = y;
      }
      u[0] = x;
}
```

Exercise 8

The purpose of this exercise is to select randomly
a character in a string without repetition like `"abcdef"`

Three main constraints :

The choice should be fair in the sense that every character
in the string should have the same probability to be chosen

The string should be read only once.

The program should not start by computing the length of the string.

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// select a random character in a string

int main() {
  int i, selected;
  char u[100]="abc";

  srand((unsigned)time(NULL));

  if (u[0] == '\0'){printf("Your string is empty");}
  else{
    selected = 0;                    // start by selection element 0
    for (i=1; u[i] != '\0' ; i++){
      if (rand() % (i + 1) == 0){    // pick a number between 0 and i
        selected=i;                  // if zero select this element i
      };
    }
    printf("The selected letter is %c at position %d\n", u[selected], selected+1);
  }
  return 0;
}
```

The scanf function

The input/output devices are treated just as files by UNIX.

In particular, among the UNIX files, one finds :

- the standard input (stdin) file number 0
- the standard output (stdout) file number 1
- the standard error (stderr) file number 2

The function **`scanf`** reads the characters from the standard input.

The function **`scanf`** consumes the characters one after the other in the order in which they were entered.

Each time the function **`scanf`** attempts to consume a character and the standard input is empty :

- the function **`scanf`** returns to the shell
- the program carries on its execution the first time the return key has been typed

The scanf function

The **`scanf`** function is not very safe but it is possible to know when it fails :

The instruction **`scanf("%d %d %d", &i, &j, &k)`** is at the same time an expression of type (signed) integer !

The value of this expression is equal to :

- **in case of success** : the number of integer variables read from the standard input
- **in case of failure :** the signed integer EOF = -1

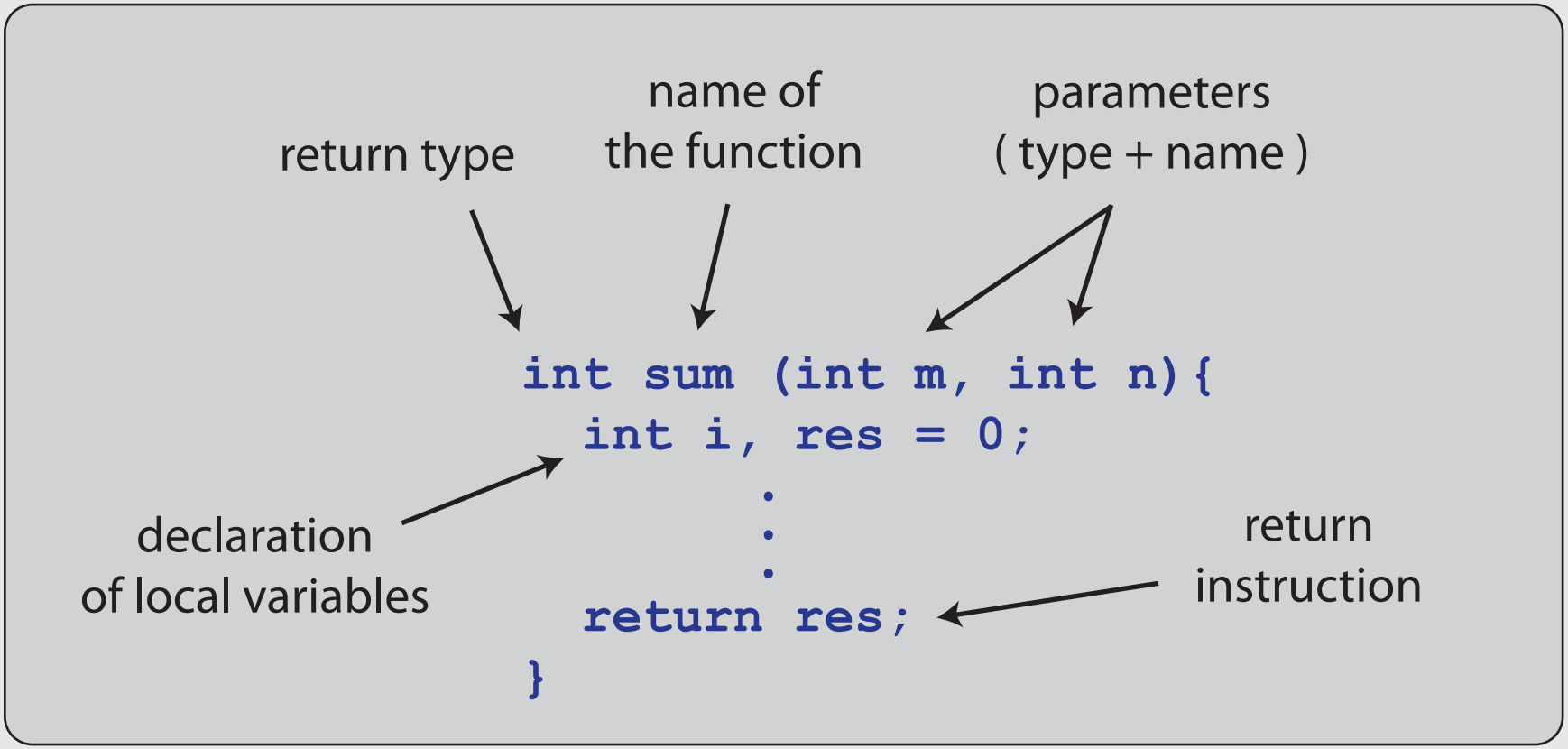
This follows a typical pattern of the programming language C : the instruction performs an operation and **at the same time** returns an integer value which indicates to the caller what happened in the course of the instruction.

Functions

How to define a function ?

```
int sum (int m, int n)   // function with two parameters
{                        // opens the block ( also called body )
                         // of the function
    int i, res = 0;
    for (i = m; i <= n; i++)
        res = res + i;   // returns m + m+1 + ... + n
    return res;
}

int main()               // main function
{
    int s;
    s = sum(10,50);      // first call of the function
    printf("the sum of numbers from 10 to 50 is equal to %d", s);
    s = sum(1,100);      // second call of the function
    printf("the sum of numbers from 1 to 100 is equal to %d", s);
    return 0;
}
```

« Anatomy » of a function



The function **sum** is called from the function **main** with integer parameters **m=10** and **n=50** the first call with integer parameters **m=1** and **n=100** the second call.

The **return** instruction enables the callee to return its result to the caller (the instruction which called it).

Call-by-value discipline

```
int swap (int x, int y){      /* WRONG */
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

```
int main(){
int a=0, b=1;
printf("a=%d and b=%d", a, b); // a=0 and b=1
swap(a, b);
printf("a=%d and b=%d", a, b); // a=0 and b=1
return 0;
}
```

Description of the call by value procedure

When the instruction `s = sum(10,50);` is reached, the program should be able to determine the value of the integer `s` .

What follows is not an exact description of the evaluation of the expression `s = sum(10,50);`
it is an idealized description which conveys the meaning (also called the semantics) of the programming language C.

When we will know more about the language, we will come back to that point, and describe the notion of « stack frame » which represents in memory the function call and its argument data.

Evaluation of : `s = sum(10,50);`

Phase 1. preparation of the function call

1. evaluation of the expressions argument of the call
   in that case : the two constant expressions of value `10` and `50`

2. allocation and initialization of the parameters
   allocation of two new variables `m` and `n`
   initialization of `m` and `n` with the values calculated in 1.

Phase 2. execution of the function call

3. execution of the body of the callee
   allocation of the local variables `i` and `res`
   the execution proceeds until it encounters the instruction `return`

Evaluation of : `s = sum(10,50);`

Phase 3. the execution encouters `return (expr);`

4. computation of the returned expression
   the value of the expression **`expr`** is computed
   this value is called the « returned value »

5. free the variables allocated for the execution
   free the variables used as parameters **`(m,n)`**
   free the variables locally declared **`(i,res)`**

6. start again the execution of the caller function
   the value determined for the expression **`sum(10,50)`**
   is equal to the « returned value » of step 4.

Local variables

The local variables of a function are :

- its parameters
- the variables declared in the body of the function

Life span of a local variable :

The local variables allocated during a function call

- are **not allocated yet** before the function call
- are **not allocated any more** after the function call

So, the variables `m,n,i,res` of the function `sum` exist in memory only during the call of the function.

Here, by local variable, we mean a local and automatic variable

Visibility rules

During the time they are allocated, the local variables of a function are only visible in the body of this function.

- From the function `main` it is impossible to access to the local variables of `sum`

  In fact, he parameters and locally declared variables of `sum` are not even allocated as long as the function `sum` is not called.

- Conversely, from the function `sum` it is impossible to access to the local variables of `main`

  The local variables of `main` remain allocated during the function call and execution of `sum` but they are not accessible.

Multiple returns

A function may have multiple returns.
This typically happens with instructions like **`if-else.`**

In that case, the **first encountered** **`return`** interrupts the execution.

```
//function max first version
int max (int m, int n)
{
  int res;
  if (m>n) {
    res = m;
  }
  else {
    res = n;
  }
  return res;
}
```

Multiple returns

A function may have multiple returns.
This typically happens with instructions like **`if-else.`**

In that case, the **first encountered** **`return`** interrupts the execution.

```
//function max second version
int max (int m, int n)
{
  if (m>n) {
    return m;
  }
  else {
    return n;
  }
}
```

Multiple returns

A function may have multiple returns.
This typically happens with instructions like **`if-else.`**

In that case, the **first encountered** **`return`** interrupts the execution.

```
//function max third version
int max (int m, int n)
{
  if (m>n) return m;
  else return n;
}
```

Multiple returns

A function may have multiple returns.
This typically happens with instructions like **`if-else.`**

In that case, the **first encountered** **`return`** interrupts the execution.

```
//function max fourth version
int max (int m, int n)
{
  return (m>n) ? m : n;
}
```

Key example

```
void print_succ(int n);
{
  n = n + 1;
  printf("%d",n);
}

int main()
{
  int i=1;
  print_succ(i);
  printf("%d",i);
  return 0;
}
```

Key example

```
void print_succ(int i);
{
  i = i + 1;
  printf("%d",i);
}

int main()
{
  int i=1;
  print_succ(i);
  printf("%d",i);
  return 0;
}
```

Functions with arrays as parameters

```
void print_content (int a[], int size){
  int i;
  for (i = 0; i < size; i++)
      printf("%d ", a[i]);
}

int main(){
  int arr[] = {0, 42, 3, 10};
  print_content(arr,4);          // print "0 42 3 10 "
  return 0;
}
```

Functions with arrays as parameters

```
int length (char s[]){
  int i = 0;
  while (s[i] != '\0')
  i++;
  return i;
}

int main(){
  char string[] = "abcdef";
  printf("%s is of length %d", string, length(string));
  return 0;
}

// content of the string : {'a','b','c','d','e','f','\0'}
// print "abcdef is of length 6"
```

Functions with arrays as parameters

```
int length (char s[]){
  int i = 0;
  for (i = 0; s[i] != '\0'; i++);
  return i;
}

int main(){
  char string[] = "abcdef";
  printf("%s is of length %d", string, length(string));
  return 0;
}

// content of the string : {'a','b','c','d','e','f','\0'}
// print "abcdef is of length 6"
```

Exercise 9

Write a function

```
int isitcorrect (char myword[], char myletters[])
```

which

- returns `1` when the string `myword` contains exactly the letters appearing in the string `myletters`
- return `0` otherwise.

Analysis of the exercise

Every letter should appear the same number of times in the two strings:

`myword` `myletters`

We typically have the choice between :

- a simple but inefficient algorithm
- a more efficient but less simple algorithm

Solution

```
int correct_word (char myword[], char myletters[])
{
  int i, inv_word[256] = {}, inv_letters[256] = {};

// construct the two inventories
  for (i = 0; myword[i] != '\0'; i++)
    inv_word[myword[i]]++;

  for (i = 0; myletters[i] != '\0'; i++)
    inv_letters[myletters[i]]++;

// verification

  for (i = 0; i < 256 ; i++){
  if (inv_word(i) != inv_letters(i))
  return 0;
  }
return 1;
```

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 !