Array and `For loop`

Pierre-Alain FOUQUE

Summary

1 - Array

2 - `for loop`

3 - Parameters on command line

Limit of basic types

- Define as many variables as memory location (a memory cell)
- Access to a variable one by one, by its name in the program

⇒ it is not possible to go a variable by its index

Array

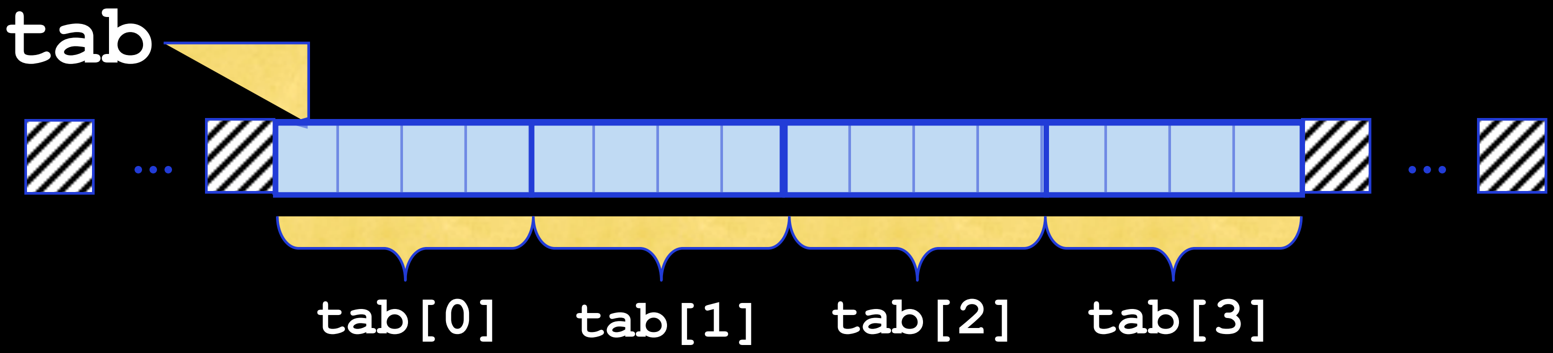
- Define under a unique name a set of consecutive memory cells of the same type

- Access to each cell by its positions (index)

The memory

```
int tab[4];
```

define an array of 4 consecutive integers



Any kind of arrays

`float tabF[10];`

`tabF`, array of 10 `floats`

`double tabD[100];`

`tabD`, array of 100 `doubles`

`char chaine[256];`

`chaine`, array of 256 characters

`int tabI[N];`

`tabI`, array of N integers

Constant size

The length of an array must be a constant

Constant size

- The length of an array must be a constant
  - either a constant « hard coded » in program (10, 100, …)

Constant size

The length of an array must be a constant

```
int tabI[20];

float tabF[10];
```

either a constant « hard coded » in program (10, 100, …)

Constant size

The length of an array must be a constant

```
int tabI[20];

float tabF[10];
```

either a constant « hard coded » in program (10, 100, ...)

either (equivalent manner !)

using the pre-processor

```
#define N 20
```

Constant size

The length of an array must be a constant

```
int tabI[20];

float tabF[10];
```

either a constant « hard coded » in program (10, 100, …)

```
#define N 10

int tabI[N];
```

either (equivalent manner !) using the pre-processor `#define N 20`

Constant size

The length of an array must be a constant

```
int tabI[20];

float tabF[10];
```

either a constant « hard coded » in program (10, 100, ...)

```
#define N 10

int tabI[N];
```

either (equivalent manner !) using the pre-processor `#define N 20`

this last manner must be use :
it is easy to change the size

Initialization of an array

During the declaration of an array, the cells contain arbitrary values :
indeed, the declaration allocates the memory and define a name, but does not affect a value into the cells !
⇒ we must initialize them, one by one.

Declaration + initialization

- Initialization : `for loop`
- Initialization cell by cell

Declaration + initialization

As for variables of basic types, it is possible to combine declaration and initialization :

```
int tab[4] = { 2, 3, -1, 5 };
```

declare the array tab of 4 integers with

```
  tab[0] = 2;
  tab[1] = 3;
  tab[2] = -1;
  tab[3] = 5;
```

Partial Initialization

We can initialize only some cells :

```
int tab[10] = { 2, , -1, 5 };
```

declare the array `tab` of 10 integers with

```
tab[0] = 2;
tab[2] = -1;
tab[3] = 5;
```

the other cells are not initialized

Array of Characters

An array of characters is a particular array : a string

`char mot[10] = “toto”;`

declare the array mot of 10 characters

`mot[0] = ‘t’;mot[1] = ‘o’;`

`mot[2] = ‘t’;mot[3] = ‘o’;`

`mot[4] = ‘\0’;` (end of string)

the other cells are not initialized

Automatic Initialization : Boucle `for`

It is possible to initialize each cell as a function of its index **`i`**

```
tabI[i] = f(i);
```

where **`f`** is a function depending on **`i`**

The instruction **`for`** allows to loop by incrementing the counter **`i`** at each round

Loop `for`

The loop **`for`** repeats an instruction many times, with a counter that increments at each round:

```
for (<init>; <test>; <incrémentation>)
  <instruction>
```

- **`<init>`** : **initialization** of the counter
- **`<test>`** : test to **continue**
- **`<incrémentation>`** :
  incrementation of the counter

for = while

The loop **for** is another formulation of the loop **while** :

for = while

The loop **for** is another formulation of the loop **while** :

```
    for (<init>; <test>; <incrémentation>)
       <instruction>
```

for = while

The loop **for** is another formulation of the loop **while** :

for

```
for (<init>; <test>; <incrémentation>)
   <instruction>
```

while

```
<init>
while (<test>){
      <instruction>
      <incrémentation>
}
```

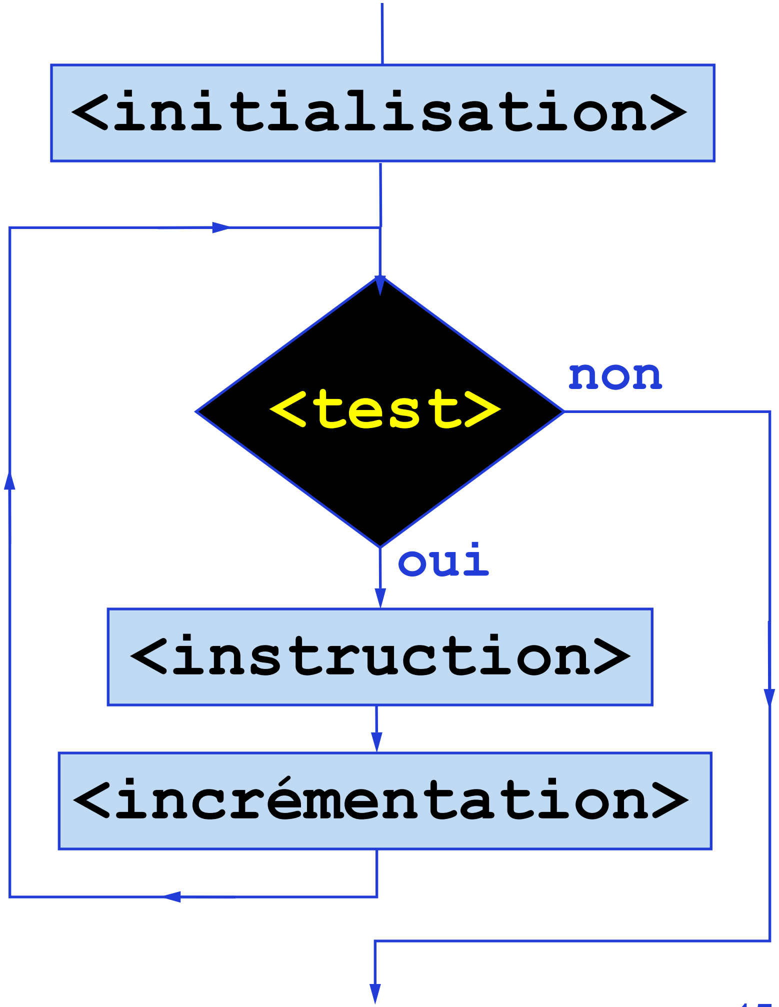
Loop `for`

Loop `for`

An instruction is executed, many times with a counter:

Loop for

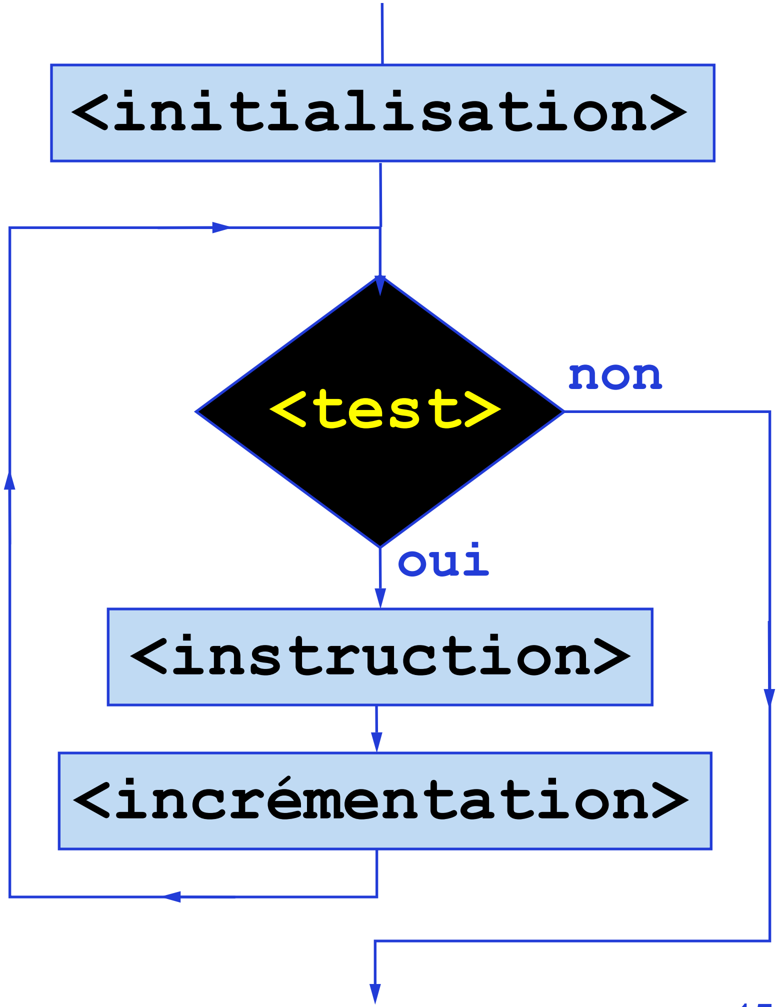
An instruction is executed, many times with a counter:



Loop for

An instruction is executed, many times with a counter:

⇒ the instruction can be never executed



`for` classique

The classical usage of the loop `for` is the following:

```
int n=15;

int i;


for (i=0; i<n; i++)
   <instruction(i)>
```

`for` classique

The classical usage of the loop `for` is the following:

```
int n=15;

int i;


for (i=0; i<n; i++)
    <instruction(i)>
```

Initialization of an array

The loop `for` allows to easily initialize an array:

```
#define N 10
int tab[N];
int i;
for (i=0; i<N; i++)
   tab[i] = 0;
```

The cells are all initialized to Zero

Initialization of an array

The loop `for` allows to initialize an array with values depending on the index of the cells:

```
#define N 10
int carre[N];
int i;
for (i=0; i<N; i++)
   carre[i] = i*i;
```

Display of an array

The function **`printf`** cannot be used to print an array, we need to print cells one by one :

```
/* carre.c - Carrés */
#include <stdio.h>

#define N 10
```

```
int main(int argc, char *argv[])
{
   int carre[N];
   int i;
   for (i=0; i<N; i++)
      carre[i] = i*i;

   for (i=0; i<N; i++)
      printf(“%d ”,carre[i]);
   printf(“\n”);
   return 0;
}
```

User Arguments

- Initializations :

```
a = 3;
b = 5;
```

⇒ values fixed at compilation

time and cannot be modified

during the execution

same value at each execution

⇒ no great interest !

```
int main()
{
   int a,b,c;
   a = 3;
   b = 5;
   c = addition(a,b);
   printf(“%d+%d=%d\n”,a,b,c);
   return 0;
}
```

argc et **argv**

```
#include <stdio.h>


int main(int argc, char *argv[])
{
   printf(“%s %s\n”,argv[0],argv[1]);
   return 0;
}
```

Function **main** :

- First argument **int argc**

  number of cells of the array
  = number of words in the command line

- Second argument **char *argv[]**

  array which contains the words on the command line

Exemple

```
#include <stdio.h>


int main(int argc, char *argv[])
{
   printf(“%s %s\n”,argv[1],argv[2]);
   return 0;
}
```

Exemple

```
#include <stdio.h>


int main(int argc, char *argv[])
{
   printf(“%s %s\n”,argv[1],argv[2]);
   return 0;
}
```

Edite ot the program

```
emacs affiche.c &
```

Compile the program

```
gcc -Wall affiche.c -o
affiche
```

Execute

```
>affiche toto tata
toto tata
```

Exemple

```
#include <stdio.h>


int main(int argc, char *argv[])
{
    printf(“%s %s\n”,argv[1],argv[2]);
    return 0;
}
```

Edite ot the program

```
emacs affiche.c &
```

Compile the program

```
gcc -Wall affiche.c -o
affiche
```

Execute

```
>affiche toto tata
toto tata
```

Indeed, the array **`char *argv[];`**

contains

`affiche` in **`argv[0]`**

`toto` in **`argv[1]`**

`tata` in **`argv[2]`**

and the integer **`int argc;`**

is equal to 3

Conversion `atoi`

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   int x;
   x = atoi(argv[1]);
   printf(“%d -> %d \n”,x,x+1);
   return 0;
}
```

Conversion `atoi`

The values of **`argv[`*`i`*`]`** are strings
⇒ we must convert them into integers
(when they code integers)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   int x;
   x = atoi(argv[1]);
   printf(“%d -> %d \n”,x,x+1);
   return 0;
}
```

Conversion `atoi`

The values of **`argv[`*`i`*`]`** are strings
⇒ we must convert them into integers
(when they code integers)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   int x;
   x = atoi(argv[1]);
   printf(“%d -> %d \n”,x,x+1);
   return 0;
}
```

```
int x;
x = atoi(argv[1]);
```

Sequences

Example : $u_i = f(i)$

Other usage : $u_i = f(i, u_{i-k}, u_{i-k+1}, \ldots, u_{i-1})$

Ex : factorial

```
/* fact.c - Factorielle */
#include <stdio.h>
#include <stdlib.h>
#define N 30
int fact[N];
```

```
int main(int argc, char *argv[])
{
   int i;
   int n = atoi(argv[1]);
   fact[0] = 1;
   for (i=1; i<=n; i++)
      fact[i] = fact[i-1]*i;
   printf(“Fact(%d) = %d \n”,n,fact[n]);
   return 0;

}
```

Several dimensions

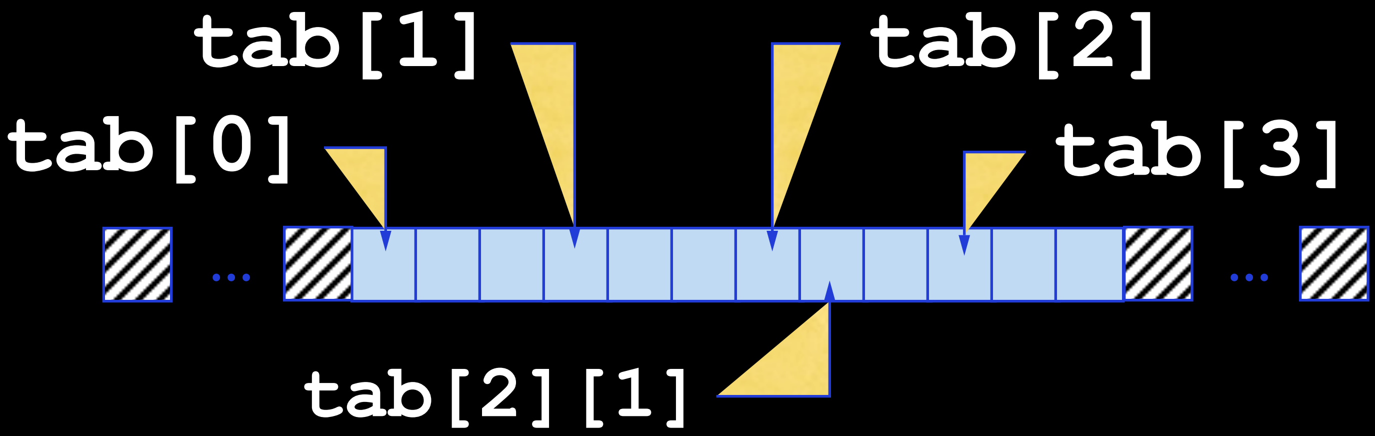
An array with many dimensions is an array of array of array of …

An array of n dimensions is an array of an array of dimension n-1

The memory

```
char tab[4][3];
```

define an array of 4 arrays of 3 characters (**`char`**) each



Loop for

```
#define M 20
#define N 10

int tab[M][N];
int i,j;

for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    tab[i][j] = 0;
```

Pointers

The pointers allow us to define array of variable size

(defined during the execution of the program)

cf. lesson on pointers and dynamic allocation