Programing in C

Pierre-Alain FOUQUE

Summary

1 - The C Language

2 - Layout of a program

3 - Data Typing

4 - Conditions

5 - Conditional loops `while`

Programming Language

The processor controls everything, but understands only the **machine language** (i. e. sequence of numbers)

- that designs operation to perform
- specific to each processor

⇒ Not easy to use / not portable (from one machine to another)

Programming Language :
interface between we and the machine

C Language

- Control Structures (if, for, while, ...)
- Usage of pointers (to access the memory)
- Iterative Programming (the program controls the changes in the memory)
- Recursive Programming (function defined by calling itself)
- Data Typing (restricted to type that can be efficiently translated into machine language)

a C Program

- Program
  - include (objects, data predefined)
  - types (new types)
  - variables (memory allocation)
  - function list
- Functions
  - header (output viewpoint)
  - operating mode : instruction list
    - simples : termined by « ; »
    - composed : simple instructions between « { ... } »

First program : « hello »

```
#include <stdio.h>


int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

First program : « hello »

inclusions

```
#include <stdio.h>


int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

First program : « hello »

inclusions

```
#include <stdio.h>


int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

First program : « hello »

inclusions

function

```
#include <stdio.h>

int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

First program : « hello »

inclusions

function

```
#include <stdio.h>


int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

First program : « hello »

inclusions

function

```
#include <stdio.h>


int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

instruction

First program : « hello »

inclusions

function

```
#include <stdio.h>


int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

instruction

First program : « hello »

inclusions

function

```
#include <stdio.h>


int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

instruction

« `main` » : main function

- only this function is called when the program is launched

⇒ distribute the tasks

Usage

```
#include <stdio.h>


int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

Usage

- Edite this program

```
xemacs hello.c &
```

```
#include <stdio.h>


int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

Usage

- Edite this program

```
xemacs hello.c &
```

- Compile this program

```
gcc -Wall hello.c -o hello
```

```
#include <stdio.h>


int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

Usage

- Edite this program

```
xemacs hello.c &
```

- Compile this program

```
gcc -Wall hello.c -o hello
```

- Execution

```
./hello
```

```
#include <stdio.h>

int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

Usage

- Edite this program

```
xemacs hello.c &
```

- Compile this program

```
gcc -Wall hello.c -o hello
```

- Execution

```
./hello
```

```
#include <stdio.h>

int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

Usage

- Edite this program

```
xemacs hello.c &
```

- Compile this program

```
gcc -Wall hello.c -o hello
```

- Execution

```
./hello
```

```
#include <stdio.h>

int main()
{
   printf("Hello World  !!\n");
   return 0;
}
```

Classical Errors

The main classical errors are :

- error in the name of a function

  ⇒ the compilater does not know it

- forget « `;` » at the end of an instruction
- use a variable not declared

  ⇒ the compilator does not know if it is an int, a float or a string !

- no « **`main`** » function

The « `main` » function

« `main` » : main function

- function called when the program is launched
- no other function is automatically executed

⇒ cannot be missed

Generic Program

```
/* Commentaires */

/* Directives pour
    le preprocessor */
#include <stdio.h>
#define TAILLE 3
#define SQ(x) x*x

/* New types */
typedef int[TAILLE] tableau;

/* Global Variables */
int globale;
```

Generic Program

Explanation :

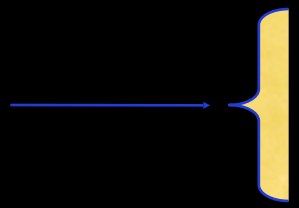
/*...*/

```
/* Commentaires */

/* Directives pour
    le preprocessor */
#include <stdio.h>
#define TAILLE 3
#define SQ(x) x*x

/* New types */
typedef int[TAILLE] tableau;

/* Global Variables */
int globale;
```

Generic Program

Explanation :
/*...*/

Preprocessor :
- inclusions
- constantes
- macros

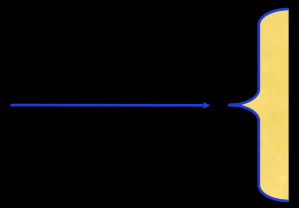


```
/* Commentaires */

/* Directives pour
    le preprocessor */
#include <stdio.h>
#define TAILLE 3
#define SQ(x) x*x

/* New types */
typedef int[TAILLE] tableau;

/* Global Variables */
int globale;
```

Generic Program

Explanation :
/*...*/

Preprocessor :
- inclusions
- constantes
- macros



Types

```
/* Commentaires */

/* Directives pour
    le preprocessor */
#include <stdio.h>
#define TAILLE 3
#define SQ(x) x*x

/* New types */
typedef int[TAILLE] tableau;

/* Global Variables */
int globale;
```

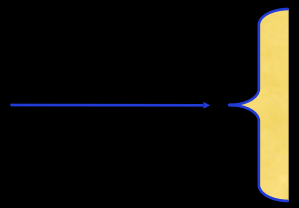
Generic Program

Explanation :
/*...*/

Preprocessor :
- inclusions
- constantes
- macros

Types

Global Variables



```
/* Commentaires */

/* Directives pour
    le preprocessor */
#include <stdio.h>
#define TAILLE 3
#define SQ(x) x*x

/* New types */
typedef int[TAILLE] tableau;

/* Global Variables */
int globale;
```

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}


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

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}


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

Header

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}


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

Header

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}


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

Header
Declaration variables

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}


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

Header
Declaration variables

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}


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

Header
Declaration variables
Instruction simple

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}


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

Header
Declaration variables
Instruction simple

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}

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

Header
Declaration variables
Instruction simple

Header

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}

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

Header
Declaration variables
Instruction simple

Header

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}

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

Header
Declaration variables
Instruction simple

Header

Declaration variables

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}

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

Header
Declaration variables
Instruction simple

Header
Declaration variables

Functions

```
int addition(int x, int y)
{
   int z;
   z = x + y;
   return z;
}

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

Header
Declaration variables
Instruction simple

Header
Declaration variables
Instructions simples

Memory

The memory stores either

- the program to execute

  (sequence of numbers in machine language for the processor)

- the variables changed by the program

$\Rightarrow$ storage by bytes or words
(blocks of 8 bits or 32 bits)

Typing

What is the coding of `01010111` ?

- the integer 87 (= 64+16+4+2+1)
- the float 0,00390625 (= 2.2^{-9})
- the character `'X'`
- an instruction in machine language
- …

⇒ we need to associate a type for each value

The typing defines the coding

Integers : `short`, `int`, `long` and `long long`

In practice, according the types and machines, the integers are coded using 8, 16, 32 or 64 bits : (GCC under Linux)

- `char` (8 bits) → +/- 127
- `short` (16 bits) → +/- 32767
- `int/long` (32 bits) → ~ +/- 2.10^9
- `long long` (64 bits) → ~ +/- 9.10^{18}
  `unsigned` int precises positive integers ⇒ we do not need a sign bit

Representating reals

- called floating (real with floating point)
- (m,e) m: mantissae and e: exponent
- rep.: Mb^e where M is the number coded in m
- base b=2, or 10 in general
- Form normalised: 1>M>=1/b
- If b=10, M>=0.1 and 3.14 is coded: (0.314,1)
- 0 cannot be represented in normalized form

Floating : `float`, `double` and `long double`

- `float` (24 + 8 bits)
  Precision 2^{-23} Min 10^{-38} Max 3.10^{38}
- `double` (53 + 11 bits)
  Precision 2^{-53} Min 2.10^{-308} Max 10^{308}
- `long double` (64 + 16 bits)
  Precision 2^{-64} Min 10^{-4931} Max 10^{4932}

Declaration and initialization

In a program or a function,
we can declare variables,
then (or simultaneously) initialize them

Declaration and initialization

In a program or a function,
we can declare variables,
then (or simultaneously) initialize them

```
#include <stdio.h>
long a = 1034827498;
float x = 1023.234;

int main()
{
   int b;
   double y; float z;
   b = 1234;
   y = 1.365; z=1.0/y;
   …
```

Declaration and initialization

In a program or a function,
we can declare variables,
then (or simultaneously) initialize them

During the declaration, the contain of the variable is random !

```
#include <stdio.h>
long a = 1034827498;
float x = 1023.234;

int main()
{
   int b;
   double y; float z;
   b = 1234;
   y = 1.365; z=1.0/y;
   …
```

Operators on numbers

The integers/floats can be manipulated thanks to the classical following operators :

- `a + b` : addition
- `a - b` : soustraction
- `a * b` : multiplication
- `a / b` : division
  (euclidean division on integers)
  (floating division on reals)
- `a % b` : modulo on integers
  (rest of the euclidean division)

Printing variables

`printf` displays on the screen, the variable contain :

`%d` for an `int` or `long` `(%3d)` `for 3 spaces (ideal for alignment)`

`%f` for a `float` or `double` `(%g)` `(%6.2f)` `for 6 spaces, 2 digits after point`

Printing variables

`printf` displays on the screen, the variable contain :

`%d` for an `int` or `long` `(%3d)` `for 3 spaces (ideal for alignment)`

`%f` for a `float` or `double` `(%g)` `(%6.2f)` `for 6 spaces, 2 digits after point`

```
…
   printf("a = %d et b = %d",a,b);
   printf("x = %f et y = %f",x,y);
   return 0;
}
```

Bitstring

- `x and y are of type unsigned int`
- `x & y` = x AND y
- `x | y` = x OR y
- `x ^ y` = x XOR y

work bit by bit.

- `x << 8` = shift of 8 bits in the left (corresponds to the multiplication by 2^8 modulo 2^{32})
- `x >> 5` = shift of 5 bits in the right
- `~x` = complement to 1 of `x`

Representation in other bases

- Decimal: Example: `1234`
- Octal: First digit is a zero. Example: `0177`
- Hexadecimal: begins by `0x` or `0X`. Ex: `0x1BF` and `0XF2A`
- To print an integer in hexadecimal form
- `int a; printf("%x",a);`

scanf

- work conversely as the **`printf function`**
- enter a value (keyboard) into a variable
- call **`scanf`** with a format and a variable which will be modified
- **`int a;`**
- **`printf("Enter a value:\n");`**
- **`scanf("%d",&a);`**

Conditionnal Execution

According the result of a test, we can hope the execution of an instruction, or not :

Conditionnal Execution

According the result of a test, we can hope the execution of an instruction, or not :

```
if (a > 0)
  printf("Positive \n");
```

Conditionnal Execution

According the result of a test, we can hope the execution of an instruction, or not :

```
if (a > 0)
  printf("Positive \n");
```

A choice:

Conditionnal Execution

According the result of a test, we can hope the execution of an instruction, or not :

```
if (a > 0)
  printf("Positive \n");
```

A choice:

```
if (a > 0)
  printf("Positive \n");
else
  printf("Négative or nul\n");
```

A test ?

The result of a test is an integer :

nul = false

non nul = truth

Operators of test

- a==b : equality test
- a!=b : difference test
- a<b or a>b : strict comparaison
- a<=b or a>=b : comparaison large

Test Combinaison

It is possible to combine (negation, conjonction, disjonction, etc) of tests

- `(!(<test>))` : negation of `<test>`
- `((<test1>) && (<test2>))`: conjonction (`<test1>` AND `<test2>`)
- `((<test1>) || (<test2>))`: disjonction (`<test1>` OR `<test2>`)

Remarks on tests

- Do not hesitate to put parenthesis
- No order is respected at execution time

  we have to see that all tests and subtests can be made so that the program can not loop!

- A single instruction is allowed after the `if` or the `else`

  if many instructions depend on the result of a test
  ⇒ composed instructions « `{...}` »

Conditional Loops

In C, it is possible to execute an instruction many times :

Loops

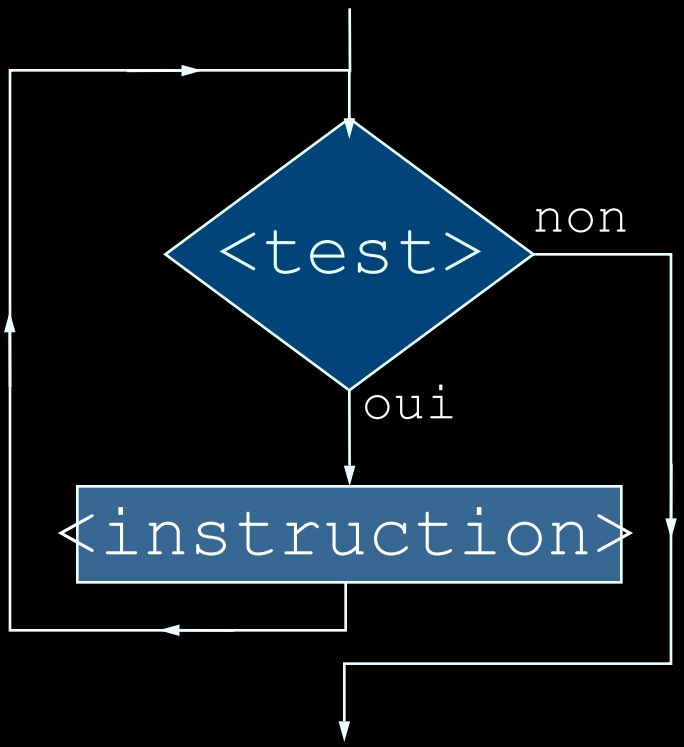
- fix number of iterations :

  `for` (cf. following lesson)

- number of iterations depending on a test : `while` and `do … while`

Loop `while`

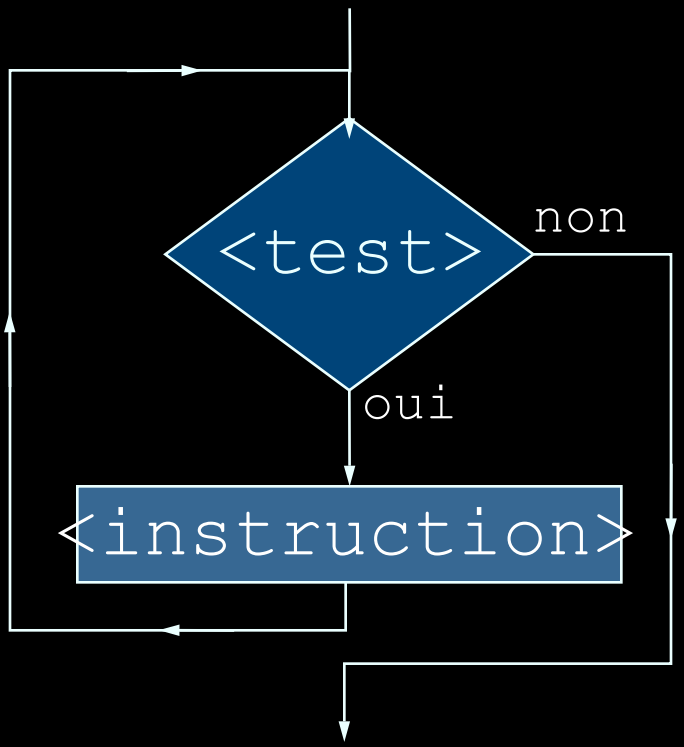
An instruction is executed while a test is satisfied :



Loop `while`

An instruction is executed while a test is satisfied :

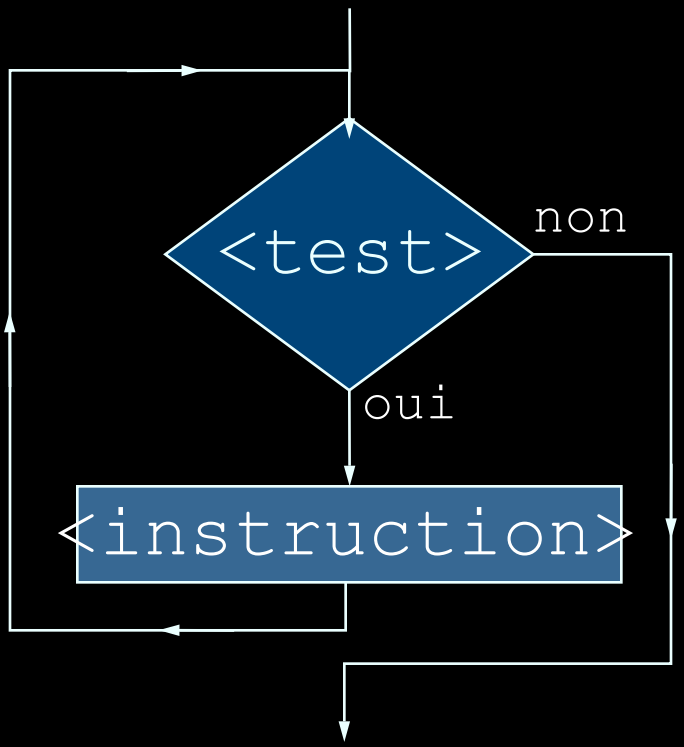
```
while <test>
```



Loop `while`

An instruction is executed while a test is satisfied :

```
while <test>
<instruction>
```

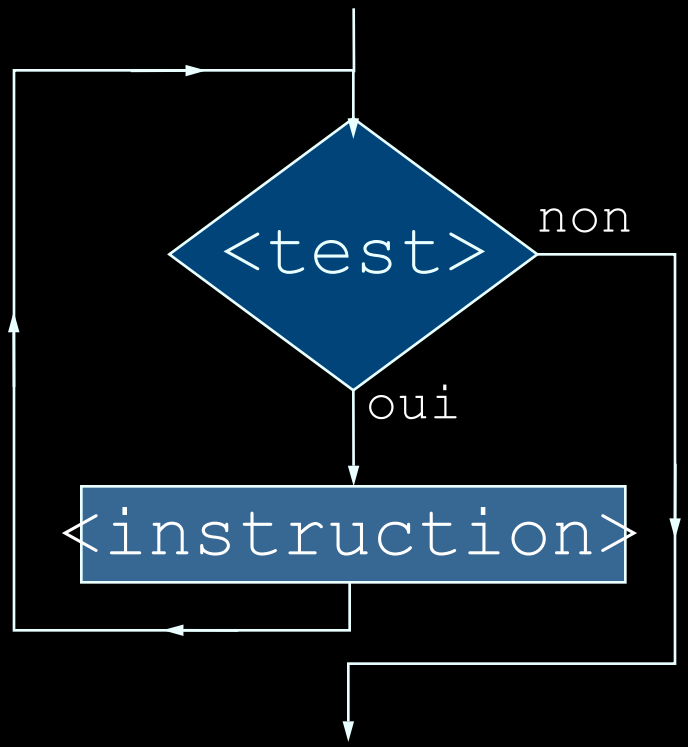


Loop `while`

An instruction is executed while a test is satisfied :

```
while <test>
<instruction>
```

⇒ instruction can be never executed



Loop `do … while`

An instruction is executed,
then repeated while a test is satisfied :

Loop `do … while`

An instruction is executed,
then repeated while a test is satisfied :

```
do
```

Loop `do … while`

An instruction is executed,
then repeated while a test is satisfied :

```
    do
<instruction>
```

Loop `do … while`

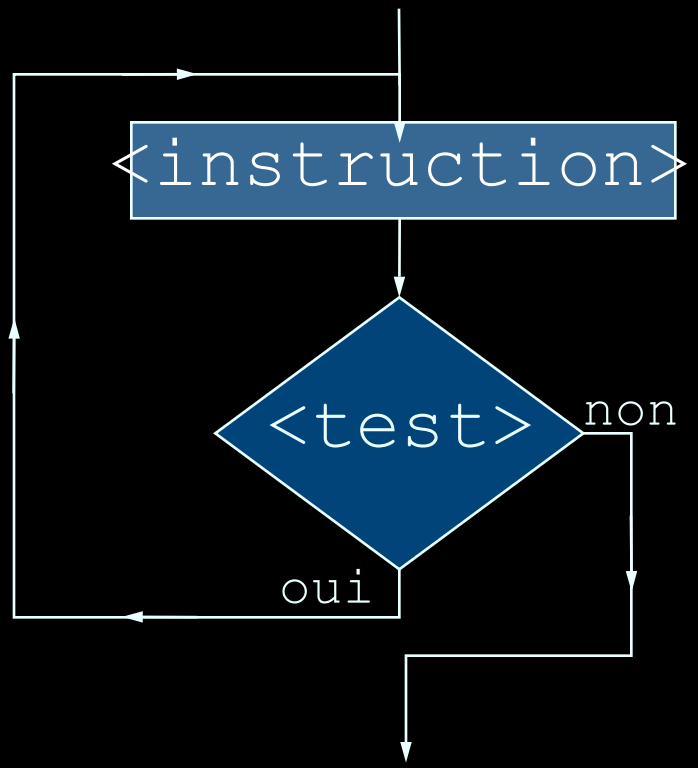
An instruction is executed,
then repeated while a test is satisfied :

```
    do
<instruction>
while <test>
```

Loop do … while

An instruction is executed,
then repeated while a test is satisfied :

```
do
<instruction>
while <test>
```

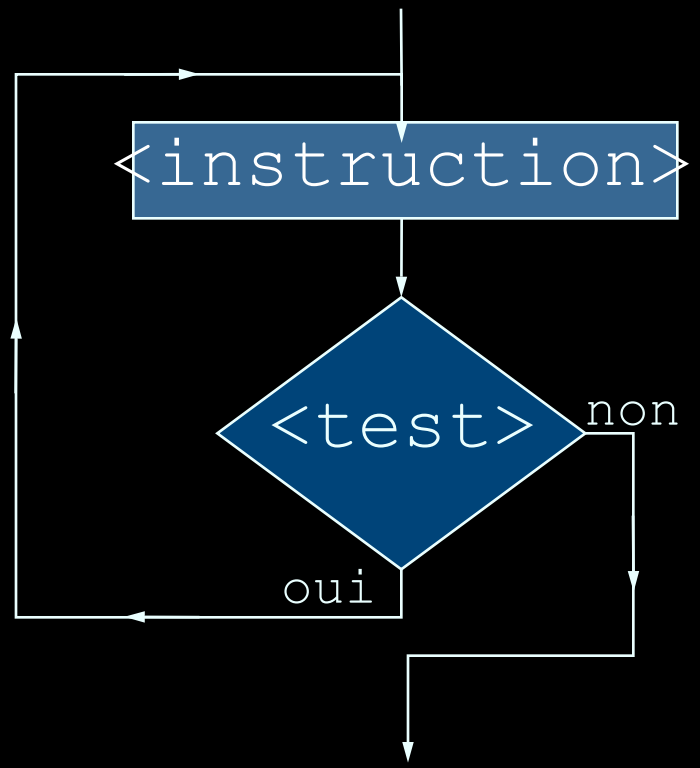


Loop `do … while`

An instruction is executed,
then repeated while a test is satisfied :

```
do
<instruction>
while <test>
```

⇒ the instruction is always executed at least one time



Remarks on the loops

- A single instruction is allowed in the loop
  - `while <test> <instruction>`
  - `do <instruction> while <test>;`

  if many instructions must be repeated
  ⇒ composed instructions « `{...}` »
- The indentation helps to see what is repeated (with the help of emacs)

Books

- Kernigham & Ritchie : Le langage C
- Bracquelaire: Méthodologie de la programmation en C
- Delannoy: Le livre du C Premier Langage

- Sedgewick : Algorithmes en C
- Cormen, Leicerson, Rivest, Stein : Introduction à l'algorithmique
- Knuth : The Art of Computer Programming