Programmation en C

Pierre-Alain FOUQUE
Département d'Informatique
École normale supérieure

Plan

1 – Le Langage C
2 – Présentation d'un programme
3 – Le typage des données
4 – Les opérateurs conditionnels
5 – Les boucles conditionnelles `while`

Langage de programmation

Le processeur contrôle tout, mais il ne comprend que le **langage machine**

soit des séries de nombres

- qui désignent l'opération à effectuer puis où l'effectuer
- spécifiques à chaque microprocesseur

⇒ Pas très facile à utiliser/pas portable

Langage de programmation : interface homme-machine

Langage C

- Structures de contrôle
- Usage des pointeurs
  pour adresser la mémoire
- Récursivité
- Typage des données

en se limitant à ce qui peut être traduit efficacement en langage machine

Un programme

- Programme = ensemble de modules
  - inclusions (objets, données prédéfinis)
  - types (nouveaux types d'objets)
  - variables (« cases » mémoires à réserver)
  - liste de fonctions
- Fonctions
  - en-tête (vue de l'extérieur)
  - mode opératoire : liste d'instructions
    - simples : terminées par « ; »
    - composées : instructions simples entre « { … } »

Premier programme : « hello »

```
#include <stdio.h>


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

Premier programme : « hello »

inclusions

```
#include <stdio.h>

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

Premier programme : « hello »

inclusions

```
#include <stdio.h>

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

Premier programme : « hello »

inclusions

fonction

```
#include <stdio.h>

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

Premier programme : « hello »

inclusions

fonction

```
#include <stdio.h>

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

Premier programme : « hello »

inclusions

fonction

```
#include <stdio.h>

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

instruction

Premier programme : « hello »

inclusions

fonction

```
#include <stdio.h>

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

instruction

Premier programme : « hello »

inclusions

fonction

```
#include <stdio.h>

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

instruction

« `main` » : fonction principale

- seule fonction appelée lors du lancement du programme

⇒ distribue les tâches

Mode d'emploi

```
#include <stdio.h>

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

Mode d'emploi

- Saisir le programme

```
xemacs hello.c &
```

```
#include <stdio.h>

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

Mode d'emploi

- Saisir le programme

```
xemacs hello.c &
```

- Compiler le programme

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

```
#include <stdio.h>

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

Mode d'emploi

- Saisir le programme

```
xemacs hello.c &
```

- Compiler le programme

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

- Exécuter

```
hello
```

```
#include <stdio.h>

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

Mode d'emploi

- Saisir le programme

```
xemacs hello.c &
```

- Compiler le programme

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

- Exécuter

```
hello
```

```
#include <stdio.h>

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

Mode d'emploi

- Saisir le programme

```
xemacs hello.c &
```

- Compiler le programme

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

- Exécuter

```
hello
```

```
#include <stdio.h>

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

Erreurs classiques

Les erreurs les plus classiques sont :

- faute dans le nom d'une fonction

  ⇒ le compilateur ne la reconnaît pas

- oubli de « `;` » en fin d'instruction simple
- utilisation d'une variable non déclarée

  ⇒ le compilateur ne sait pas si c'est un entier, un réel ou une chaîne de caractères !

- pas de fonction « **`main`** »

Fonction « `main` »

« **`main`** » : fonction principale

- fonction appelée lors du lancement du programme
- aucune autre n'est exécutée automatiquement

⇒ indispensable

Programme générique

```
/* Commentaires */

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

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

/* Variables globales */
int globale;
```

Programme générique

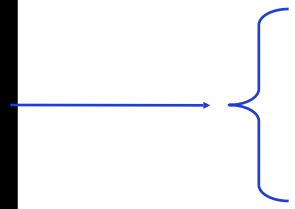
Commentaires :
à tout moment,
entre /*...*/

```
/* Commentaires */

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

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

/* Variables globales */
int globale;
```

Programme générique

Commentaires :
à tout moment,
entre /*...*/

Préprocesseur :
- inclusions
- constantes
- macros



```
/* Commentaires */

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

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

/* Variables globales */
int globale;
```

Programme générique

Commentaires :
à tout moment,
entre /*...*/

Préprocesseur :
- inclusions
- constantes
- macros

Types

```
/* Commentaires */

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

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

/* Variables globales */
int globale;
```

Programme générique

Commentaires :
à tout moment,
entre /*...*/

Préprocesseur :
- inclusions
- constantes
- macros

Types

Variables globales

```
/* Commentaires */

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

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

/* Variables globales */
int globale;
```

Macros

ATTENTION:

- `#define SQ(a) a*a`
- Si on écrit `SQ(a+b)` : on va obtenir `a+b*a+b` ≠ `(a+b)*(a+b)`

⇒`#define SQ(a) ((a)*(a))`

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

Programme générique (suite) fonctions

```
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;
}
```

Programme générique (suite) fonctions

```
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;
}
```

En-tête

Programme générique (suite) fonctions

```
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;
}
```

En-tête

Programme générique (suite) fonctions

```
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;
}
```

En-tête

Déclaration variables

Programme générique (suite) fonctions

```
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;
}
```

En-tête

Déclaration variables

Programme générique (suite) fonctions

```
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;
}
```

En-tête
Déclaration variables
Instruction simple

Programme générique (suite) fonctions

```
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;
}
```

En-tête
Déclaration variables
Instruction simple

Programme générique (suite) fonctions

```
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;
}
```

En-tête
Déclaration variables
Instruction simple

En-tête

Programme générique (suite) fonctions

```
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;
}
```

En-tête
Déclaration variables
Instruction simple

En-tête

Programme générique (suite) fonctions

```
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;
}
```

En-tête
Déclaration variables
Instruction simple

En-tête

Déclaration variables

Programme générique (suite) fonctions

```
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;
}
```

En-tête
Déclaration variables
Instruction simple

En-tête

Déclaration variables

Programme générique (suite) fonctions

```
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;
}
```

En-tête
Déclaration variables
Instruction simple

En-tête

Déclaration variables

Instructions simples

Mémoire

La mémoire stocke indifféremment

- le programme à exécuter

  (série de codes en langage machine adressés au processeur)

- les variables manipulées par le programme

⇒ stockage par octets ou mots

(blocs de 8 bits ou 32 bits)

Typage

Mais que code `01010111` ?

- L'entier 87 (= 64+16+4+2+1)
- Le flottant 0,00390625 (= 2.2^{-9})
- Le caractère `'x'`
- Une instruction en langage machine
- …

⇒ il faut associer un type à chaque valeur

Le typage définit le codage

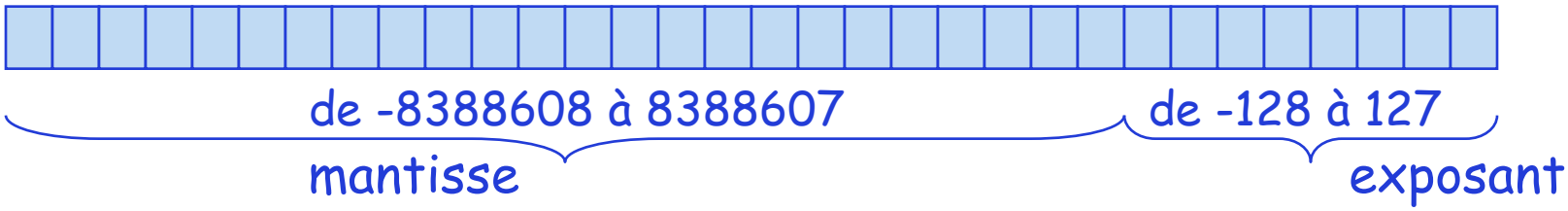
Entiers : `short`, `int`, `long` et `long long`

En pratique, selon les types et machines, les entiers sont codés sur 8, 16, 32 ou 64 bits : (GCC sous Linux)

- **`short`** (16 bits) → +/- 32767
- **`int/long`** (32 bits) → ~ +/- 2.10^9
- **`long long`** (64 bits) → ~ +/- 9.10^{18}

Le qualitatif **`unsigned`** précise que les entiers seront positifs
⇒ plus besoin du bit de signe

Flottants : `float`, `double` et `long double`



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

Déclaration et initialisation

Dans un programme ou une fonction,
on peut déclarer des variables,
puis (ou simultanément) les initialiser

Déclaration et initialisation

Dans un programme ou une fonction,
on peut déclarer des variables,
puis (ou simultanément) les initialiser

```
#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;
   …
```

Déclaration et initialisation

Dans un programme ou une fonction,
on peut déclarer des variables,
puis (ou simultanément) les initialiser

Lors de la déclaration, le contenu de la variable est aléatoire !

```
#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;
   …
```

Opérateurs sur les nombres

Les entiers/flottants peuvent être manipulés grâce aux opérateurs classiques suivants :

- **`a + b`** : addition
- **`a - b`** : soustraction
- **`a * b`** : multiplication
- **`a / b`** : division
  (division euclidienne sur les entiers)
  (division flottante sur les réels)
- **`a % b`** : modulo sur les entiers
  (reste de la division euclidienne)

Affichage des variables

`printf` affiche sur la sortie standard (écran) le contenu de variables :

- **`%d`** pour un **`int`** ou **`long`**
- **`%f`** pour un **`float`** ou **`double`**

Affichage des variables

`printf` affiche sur la sortie standard (écran) le contenu de variables :

- **`%d`** pour un **`int`** ou **`long`**
- **`%f`** pour un **`float`** ou **`double`**

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

Chaînes de bits

- **`x & y = x AND y`**
- **`x | y = x OR y`**
- **`x ^ y = x XOR y`**

opère bit à bit.

- **`x << 8`** = décalage de 8 bits vers la gauche (correspond à la multiplication par 2**8 modulo 2**32)
- **`x >> 5`** = décalage de 5 bits vers la droite
- **`~x`** = complément à 1 de **`x`**

Représentation dans d'autre

- Décimale: Exemple: **`1234`**
- Octale: Premier chiffre est un zéro. Exemple: **`0177`**
- Hexadécimale: commence par **`0x`** ou **`0X`**. Ex: **`0x1BF`** et **`0XF2A`**
- Pour imprimer un entier sous forme hexadécimal
- **`int a; printf("%x",a);`**

scanf

- fonctionne à l'inverse de la fonction **`printf`**
- entrer une valeur dans une variable
- appeler **`scanf`** avec un format et une variable avec modification de la variable
- **`int a;`**
- **`printf("Entrer une valeur: ");`**
- **`scanf("%d",&a);`**

Exécution conditionnelle

En fonction du résultat d'un test, on peut souhaiter exécuter une instruction, ou pas :

Exécution conditionnelle

En fonction du résultat d'un test, on peut souhaiter exécuter une instruction, ou pas :

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

Exécution conditionnelle

En fonction du résultat d'un test, on peut souhaiter exécuter une instruction, ou pas :

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

Une alternative:

Exécution conditionnelle

En fonction du résultat d'un test, on peut souhaiter exécuter une instruction, ou pas :

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

Une alternative:

```
if (a > 0)
  printf("Positif \n");
else
  printf("Négatif ou nul\n");
```

Un test ?

Le résultat d'un test est un entier :

nul = faux

non nul = vrai

Opérateurs de test

- a==b : test d'égalité
- a!=b : test de différence
- a<b ou a>b : comparaison stricte
- a<=b ou a>=b : comparaison large

Combinaison de tests

Il est possible de combiner (négation, conjonction, disjonction, etc) des tests

- `(!(<test>))` : négation de `<test>`
- `((<test1>) && (<test2>))` : conjonction (`<test1>` ET `<test2>`)
- `((<test1>) || (<test2>))` : disjonction (`<test1>` OU `<test2>`)

Remarques sur les tests

- Ne pas hésiter à mettre des parenthèses
- Aucun ordre n'est respecté à l'exécution

  il faut donc veiller à ce que tous les tests et sous-tests puissent être effectués sans faire « planter » le programme !

- Une seule instruction est permise après le `if` ou le `else`

  si plusieurs instructions sont conditionnées par le résultat du test
  ⇒ instructions composées « `{…}` »

Boucles conditionnelles

En C, il est possible de faire répéter une instruction un grand nombre de fois :

boucles

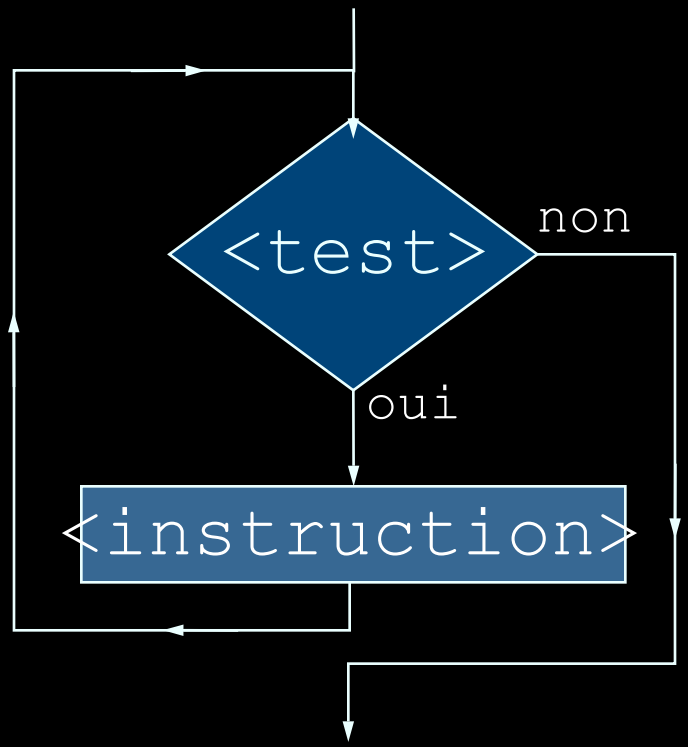
- nombre d'itérations fixé :

  `for` (cf. cours suivant)

- nombre d'itérations dépendant d'un test : `while` **et** `do … while`

Boucle `while`

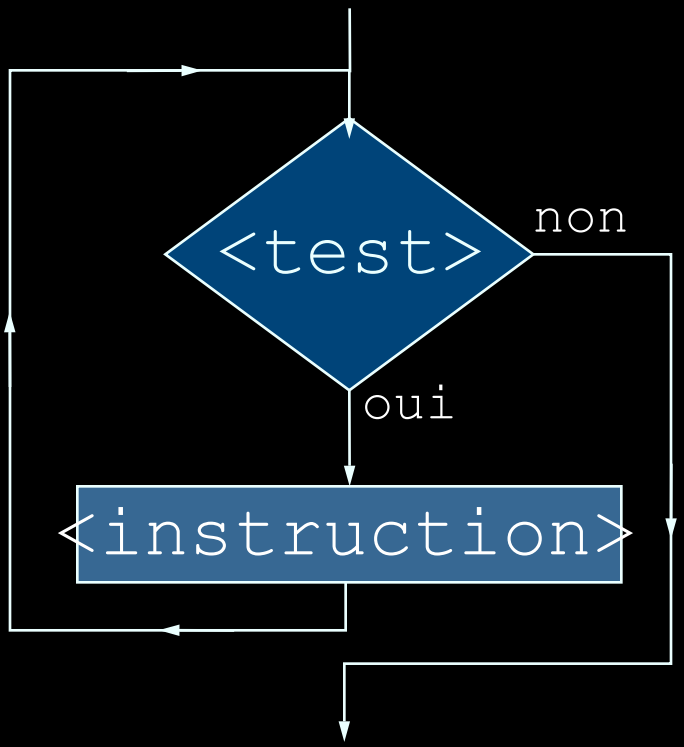
Une instruction est exécutée,
tant qu'un test est satisfait :



Boucle `while`

Une instruction est exécutée,
tant qu'un test est satisfait :

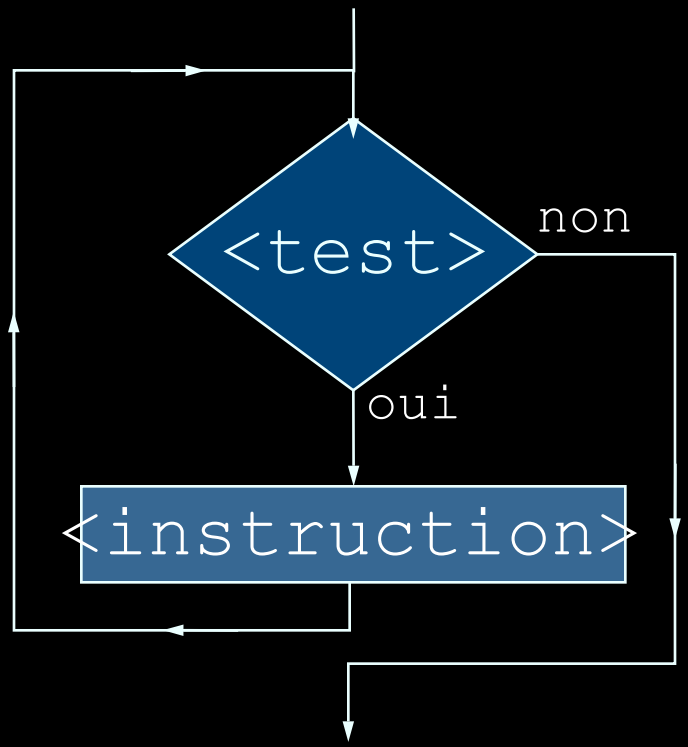
```
while <test>
```



Boucle `while`

Une instruction est exécutée,
tant qu'un test est satisfait :

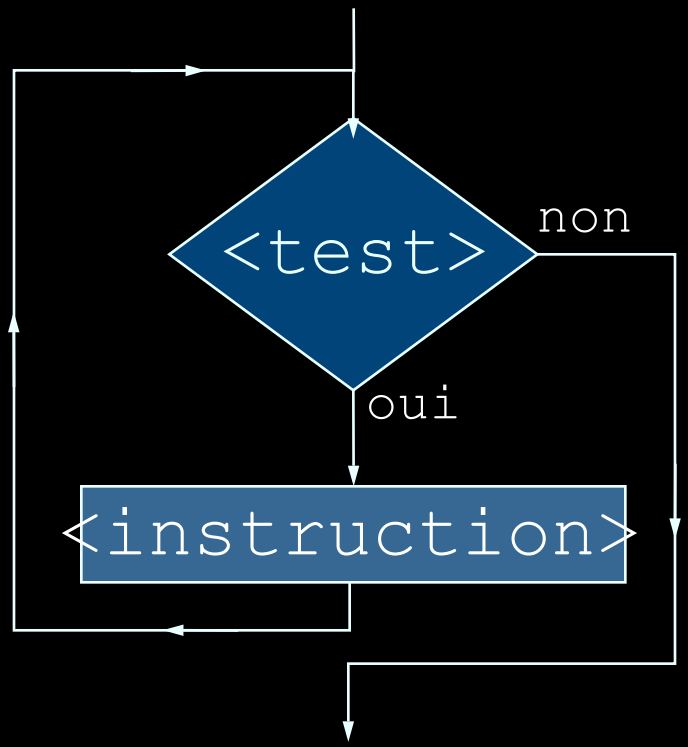
```
while <test>
<instruction>
```



Boucle `while`

Une instruction est exécutée,
tant qu'un test est satisfait :

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



⇒ l'instruction peut ne jamais être exécutée

Boucle `do … while`

Une instruction est exécutée,
puis répétée tant qu'un test est satisfait :

Boucle `do … while`

Une instruction est exécutée,
puis répétée tant qu'un test est satisfait :

```
do
```

Boucle `do … while`

Une instruction est exécutée,
puis répétée tant qu'un test est satisfait :

```
    do
<instruction>
```

Boucle `do … while`

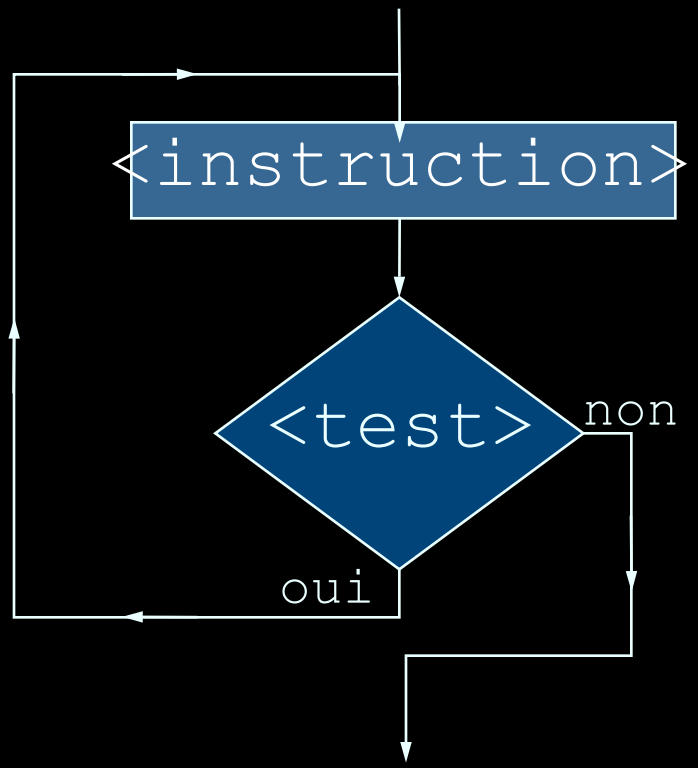
Une instruction est exécutée,
puis répétée tant qu'un test est satisfait :

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

Boucle `do … while`

Une instruction est exécutée,
puis répétée tant qu'un test est satisfait :

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

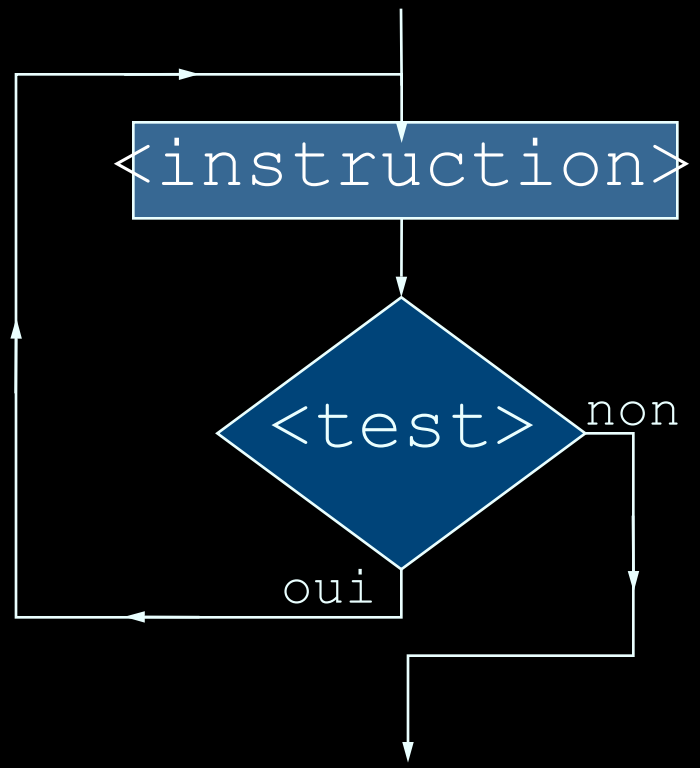


Boucle `do … while`

Une instruction est exécutée,
puis répétée tant qu'un test est satisfait :

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

⇒ l'instruction est toujours exécutée au moins une fois



Remarques sur les boucles

- Une seule instruction est permise dans la boucle
  - `while <test> <instruction>`
  - `do <instruction> while <test>;`

  si plusieurs instructions doivent être répétées
  ⇒ instructions composées « `{…}` »
- L'indentation aide à repérer ce qui est répété (avec l'aide d'emacs également)

Ouvrages de référence

- Kernigham & Ritchie : Le langage C
- Bracquelaire: Méthodologie de la programmation en C

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