Introduction à l'algorithmique et au langage C

Introduction aux langages de programmation (1/2)

L. Beaudoin & R. Erra & A. Gademer & L. Avanthey

`http://learning.esiea.fr/`

INF1031 - LAB1413, 2016 - 2017

Plan

Plan: Introduction

Objectifs de cette présentation

- Notion de programme informatique
- Langages de programmation et compilation
- Premier programme en C

Plan: Programmer

Définition : Programmer

- Programmer :
  - Théorie : « Fournir des instructions à la machine pour qu'elle réalise une tâche. »
  - Pratique : « Écrire une liste ordonnée d'instructions processeur qui retranscrit l'algorithme permettant la réalisation de la tâche. »

- Cette **liste ordonnée d'instructions** quand elle est effectuée par un ordinateur s'appelle un **programme**.

- Comment écrit-on ce programme ?

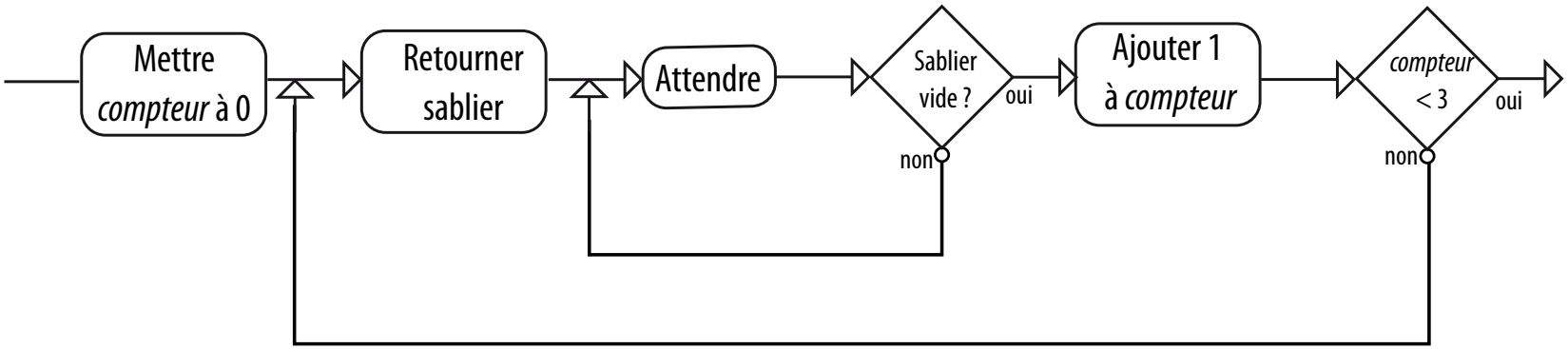
  $\Rightarrow$ Grâce à un **langage de programmation**.

Étapes de construction d'un programme

1. Comprendre le problème
2. Prendre des exemples simples pour construire une solution à partir des intuitions
3. Faire le diagramme de flux
4. Écrire le pseudo-code
5. Portage sur l'ordinateur : convertir le pseudo en langage de programmation

Diagramme de flux (Flowchart)

- Description dans un langage graphique normalisé des étapes de l'algorithme



- Avantage : Le déroulement de la séquence est très facile à lire
- Inconvénient : les machines n'aiment pas lire les dessins

Pseudo-code

- Pseudo-code : description en « langage naturel » des étapes de l'algorithme

Mettre `compteur` à 0
répéter
  Retourner sablier
  répéter
    Attendre
  tant que Sablier non vide
  Ajouter 1 à `compteur` et mémoriser
tant que `compteur` inférieur strict à 3

- Avantage : Formel. Facile à comprendre par un humain
- Inconvénient : beaucoup d'implicites (qu'est ce qu'un sablier ? comment sait-on qu'il est vide ? que veut dire retourner ?)

Plan: Langages

Définition : Langages de programmation

- Langage de programmation : « Langage permettant de décrire la suite ordonnée d'instructions que l'on désire voir exécutées par la machine. »

- On trouve de nombreux langages de programmation, mais l'un d'entre eux est essentiel : on l'appelle le **langage machine**.

Langage machine

- Ce langage est **uniquement constitué de nombres** (identifiant du type d'instruction, adresse du registre, valeur numérique) et il est **le seul qui soit compris par le processeur** (c'est le contenu brut de la pile d'instructions).
- Les **fichiers binaires** (fichiers dits exécutables par abus de langage) sont des programmes en langage machine
- Le langage machine a deux défauts : il est strictement **dépendant du type de processeur** (x86, ARM, DsPic...) et il est **difficilement lisible** par un humain.

```
0000 426F6E6A    # Definition en memoire de la chaine a afficher.
0008 B8040000    # Mettre 4 dans le reg. eax ("Write")
000d BB010000    # Mettre 1 dans le reg. ebx ("STDOUT")
0012 B9000000    # Mettre l'adresse mem. de notre chaine dans le reg. ecx
0017 BA080000    # Mettre la taille de la chaine dans edx
001c CD80        # Interruption 0x80 (appel systeme sous Linux)
001e B8010000    # Mettre 1 dans eax ("Exit")
0023 BB000000    # Mettre 0 dans ebx (valeur de retour)
0028 CD80        # Interruption 0x80 (appel systeme sous Linux)
```

Langage assembleur

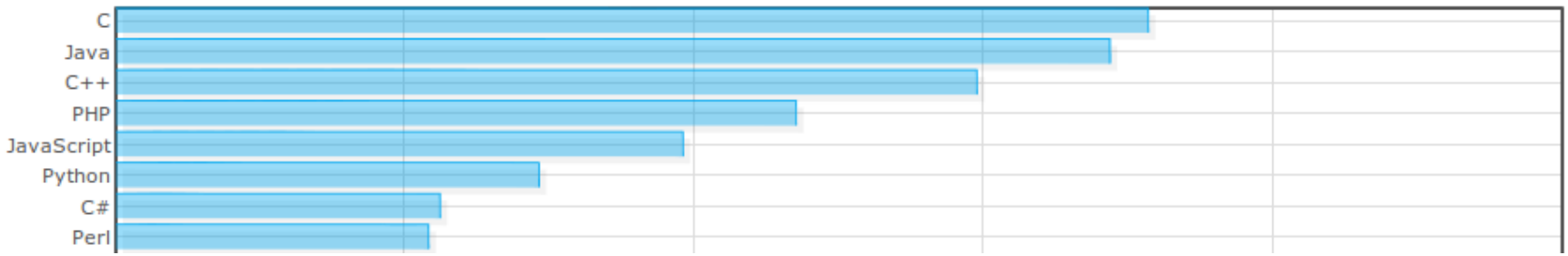
- Ce langage est une **transcription textuelle lisible** des instructions du langage machine. La forme des instructions est identique au langage machine (identifiant du type d'instruction, adresse du registre, valeur numérique) et comme celui-ci le langage assembleur reste strictement **dépendant du type de processeur** (x86, ARM, DsPic...).
- Il reste encore **très utilisé** car il permet un **contrôle total** de ce qui est effectué par le processeur.

```
BONJ:   .ascii  "Bonjour\n"     # Definition en memoire de la chaine a afficher.
_start: mov     $4      , %eax  # Mettre 4 dans le reg. eax ("Write")
        mov     $1      , %ebx  # Mettre 1 dans le reg. ebx ("STDOUT")
        mov     $BONJ   , %ecx  # Mettre l'adresse mem. de notre chaine dans le reg. ecx
        mov     $8      , %edx  # Mettre la taille de la chaine dans edx
        int     $0x80           # Interruption 0x80, (appel systeme sous Linux)
        mov     $1      , %eax  # Mettre 1 dans eax ("Exit")
        mov     $0      , %ebx  # Mettre 0 dans ebx (valeur de retour)
        int     $0x80           # Interruption 0x80, (appel systeme sous Linux)
```

Langages

- Afin de faciliter leur travail, les programmeurs ont inventés d'autres langages : FORTRAN, PASCAL, C, C++, Bash, Java, Python, Ruby, etc.
- Syntaxe plus facilement manipulable par des humains (langage de bas/haut niveau d'abstraction)
- Néanmoins ces derniers nécessitent d'être traduits à la machine, soit à l'avance (compilateur), soit au fur et à mesure (interpréteur).

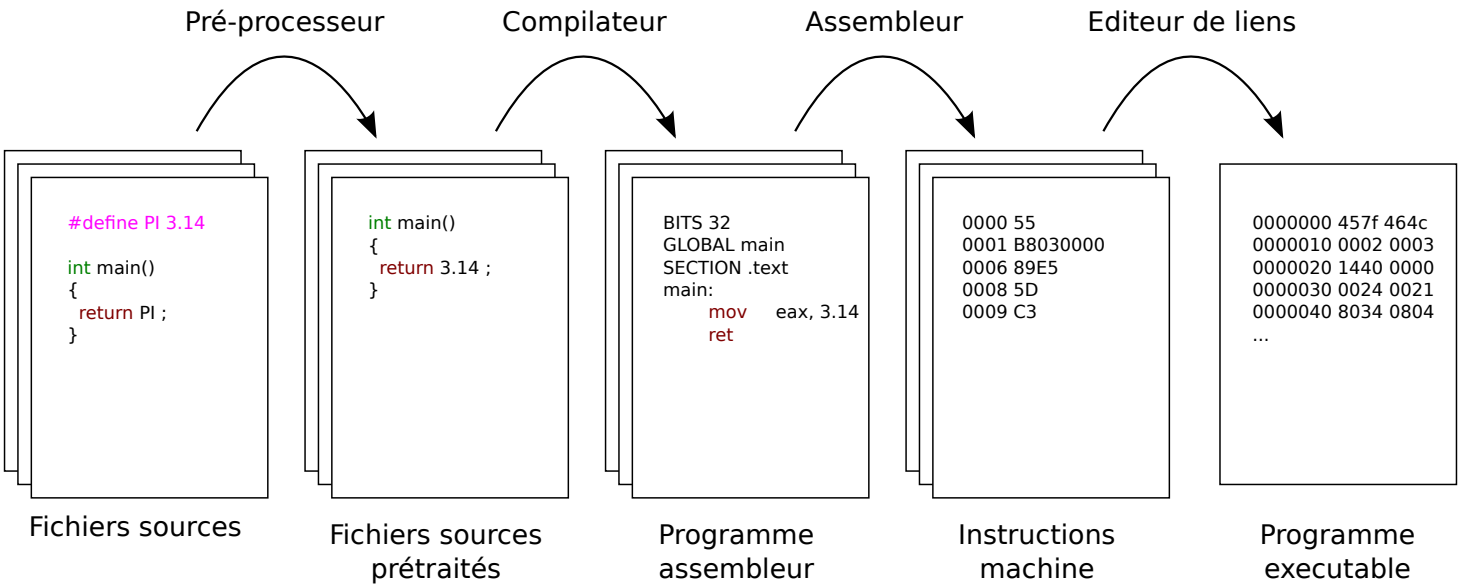
Langages les plus utilisés (2011) `http://langpop.com/`



(À voir) Genèse des langages : `http://oreilly.com/news/graphics/prog_lang_poster.pdf`

Compilation

- Tous les langages de programmation nécessitent une phase de **traduction en langage machine** pour être utilisés par le processeur. Si cette phase s'effectue avant l'exécution, on l'appelle la **compilation** et elle est effectuée à l'aide d'un programme appelé **compilateur**.



Compilation

- Le compilateur utilise sa connaissance de la **syntaxe** (orthographe des mots clefs) et de la **grammaire** (règles d'agencement des mots) du langage pour traduire les instructions d'un langage à l'autre.
- Les **fautes de syntaxe ou de grammaire** empêchent la compilation d'être menée à bien : ce sont les **erreurs de compilation**.

- On remarquera que le compilateur, en tant que programme, a lui-même dû être compilé auparavant ... ? ? ?

Langages

- Les langages comme le C, le Java ou le Python définissent leur **propres règles de syntaxe et de grammaire**, mais reposent sur des **concepts similaires** tirés des principes de l'algorithmique : embranchements, boucles et fonctions.
- C'est pourquoi il est **facile d'apprendre de nombreux langages** de programmation différents.

Langages

- On distingue cependant plusieurs **familles de langage** dont l'approche des problèmes est différente :
  - les langages de programmation procédurale/impérative (vue au cours du cycle préparatoire et utilisée durant les 5 années)
  - les langages de programmation objet (dont les concepts sont abordés au cours du cycle ingénieur).
- Ce module se concentrera sur les principes de base de l'**algorithmique**, les langages de **programmation procédurale** et plus particulièrement le **langage C**.

Plan: Code source

Exécutable & Code source

- Un exécutable est un **fichier binaire** (composé de 0/1) correspondant aux instructions du programme en langage machine.
- Il peut être obtenu par compilation d'un **code source** écrit en langage C.
- Un code source est un **fichier texte** respectant la grammaire et la syntaxe du langage de programmation.
- Un code source C possède une extension `.c`

Code minimal

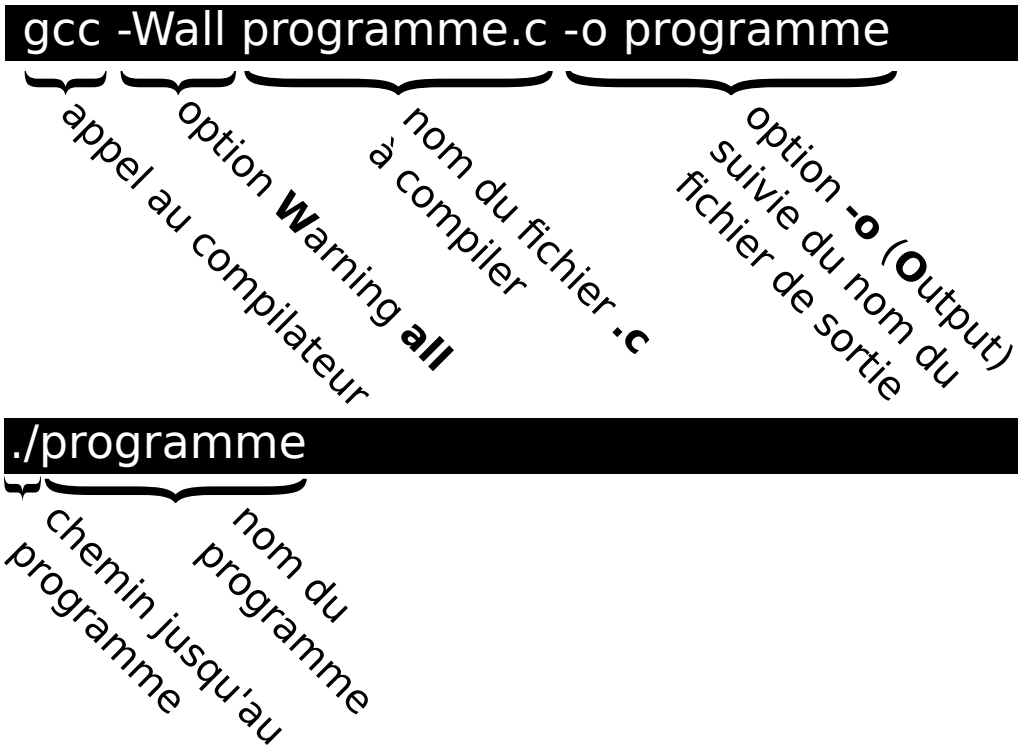
Programme C minimal :

```
/* LES DEFINITIONS (Facultatif) */

int main() { // Fonction principale

   /* PARTIE DECLARATIVE */

   /* PARTIE EXECUTIVE */

   return 0; // Fin du programme
}
```

- NB1 : Ce qui est situé entre `/* */` ou les lignes commençant par `//` $\Rightarrow$ **commentaires** non lus par le compilateur
- NB2 : les lignes d'instructions **finisse toujours par ;**

Compilation

- Il existe plusieurs compilateurs pour le langage C
- Nous utiliserons celui qui s'appelle GCC (GNU Compiler Collection)



Compilation avancée

1. Étape pré-processeur

```
gcc -E prog.c -o prog.i
```

2. Étape compilateur

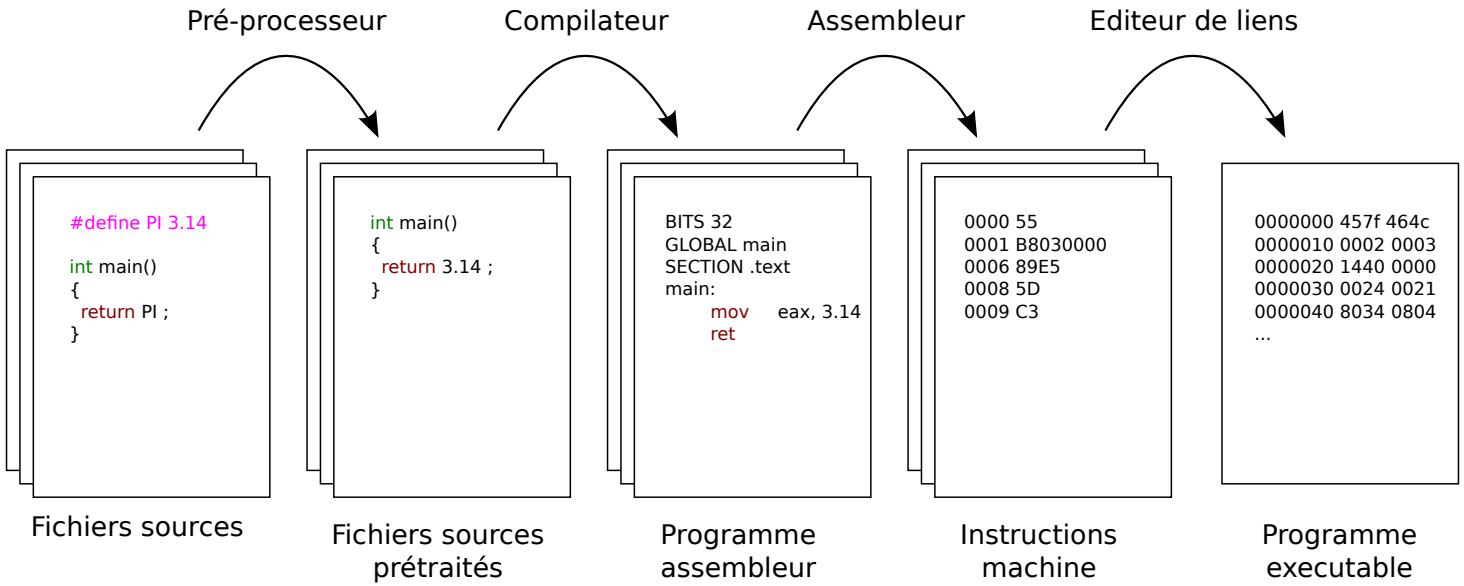
```
gcc -S prog.i -o prog.s
```

3. Jusqu'à l'étape assembleur

```
gcc -c prog.c -o prog.o
```

4. Étape édition de lien

```
gcc prog.o -o prog
```



Rappels

Un programme est l'implémentation d'un algorithme, donc :

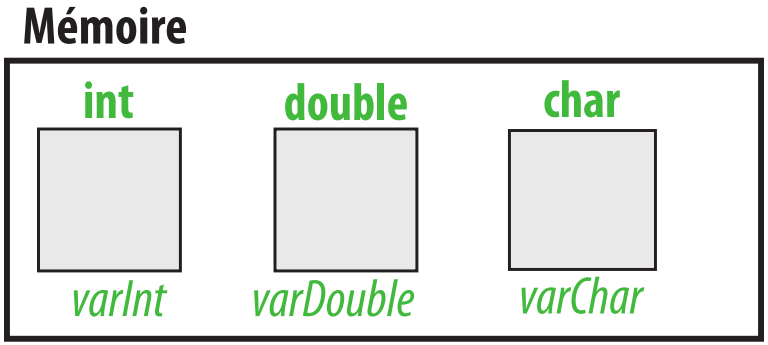
- Un programme se déroule de manière **séquentielle** : une ligne après l'autre
- Un programme peut **tester** la valeur d'une variable ou d'un registre (variable liée au monde extérieur)
- L'exécution d'une partie du programme peut être **conditionnelle** $\Rightarrow$ Embranchements
- Certains tests permettent de revenir à une ligne antérieure, afin de **répéter** une partie du programme $\Rightarrow$ Boucles
- Les variables servent à **mémoriser** des informations et des **calculs**

Plan: Variables et opérateurs

Variable et registres

- Ordinateur de bureau : seul le Système d'Exploitation accède au matériel via les drivers : pas d'accès direct aux registres mémoire (vs. micro-contrôleur) $\Rightarrow$ variables.
- Le **nom** d'une variable permet d'accéder à l'espace de la mémoire vive réservé.
- Un ordinateur ne manipule que des 0/1 : un entier, un flottant et un texte sont donc indiscernables sauf par le **type** de la variable : `int` (entier relatif), `double` (flottant), `char` (caractère).

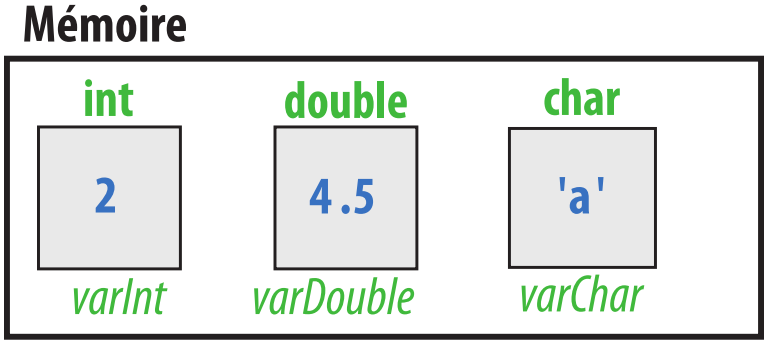
```
int varInt;
double varDouble;
char varChar;
```



Affectation

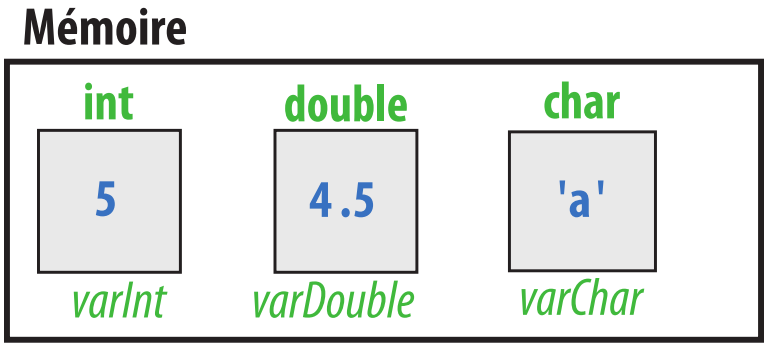
- Opérateur d'affectation =
- Permet de mémoriser l'information

```
varInt = 2;
varDouble = 4.5;
varChar = 'a';
```



- Permet de mémoriser des résultats

```
varInt = varInt + 3;
```



Opérateurs mathématiques

- Opérateur mathématique : +, -, *, /, % (modulo → reste de la division),

```
varInt = 4 * 2;     // varInt vaut ?
varInt = 7 / 5;     // varInt vaut ?
varInt = 7 % 5;     // varInt vaut ?
varDouble = 7 / 5; // varDouble vaut ?
varDouble = 7.0 / 5.0; // varDouble vaut ?
```

- Incrémentation : ++, --, +=, -=, *=, /=, %=,

```
varInt = 2;  // varInt vaut ?
varInt++;    // varInt vaut ?
varInt += 3; // varInt vaut ?
varInt *= 2; // varInt vaut ?
```

Opérateurs mathématiques

- Opérateur mathématique : +, -, *, /, % (modulo → reste de la division),

```
varInt = 4 * 2;     // varInt vaut 8
varInt = 7 / 5;     // varInt vaut 1 !
varInt = 7 % 5;     // varInt vaut 2
varDouble = 7 / 5; // varDouble vaut 1.0
varDouble = 7.0 / 5.0; // varDouble vaut 1.2
```

- Incrémentation : ++, --, +=, -=, *=, /=, %=,

```
varInt = 2;  // varInt vaut 2
varInt++;    // varInt vaut 3
varInt += 3; // varInt vaut 6
varInt *= 2; // varInt vaut 12
```

Opérateurs de tests

- Booléen : vrai (1) ou faux (0)
- Tests : == (égal), != (différent), <=, <, >, >=,
- Conjonction logique : && (et), || (ou), ! (non),

```
varInteger = 2;
(varInteger == 2) // ?
(varInteger != 4) // ?
(varInteger > 3)  // ?
(varInteger <= 2) // ?
((varInteger > 0)&&(varInteger > 4))      // ?
((varInteger == 0)||(!(varInteger > 4))) // ?
```

Opérateurs de tests

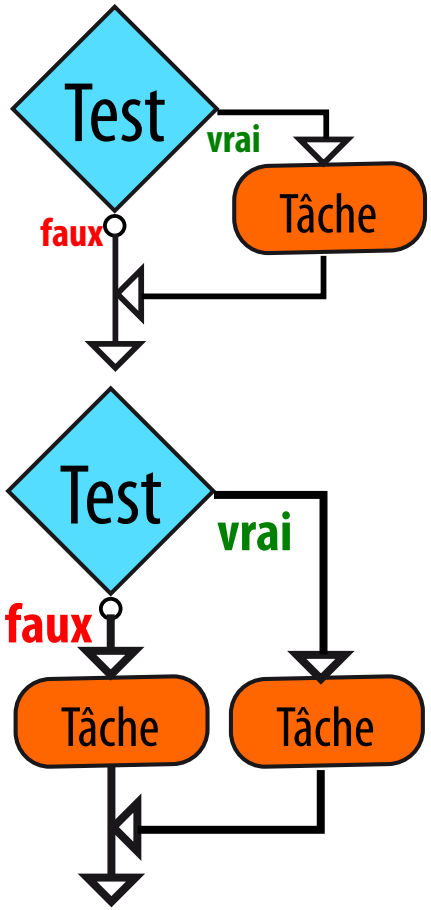
- Booléen : vrai (1) ou faux (0)
- Tests : == (égal), != (différent), <=, <, >, >=,
- Conjonction logique : && (et), || (ou), ! (non),

```
varInteger = 2;
(varInteger == 2) // Vrai
(varInteger != 4) // Vrai
(varInteger > 3)  // Faux
(varInteger <= 2) // Vrai
((varInteger > 0)&&(varInteger > 4))      // Faux
((varInteger == 0)||(!(varInteger > 4))) // Vrai
```

Plan: Embranchements

Embranchements

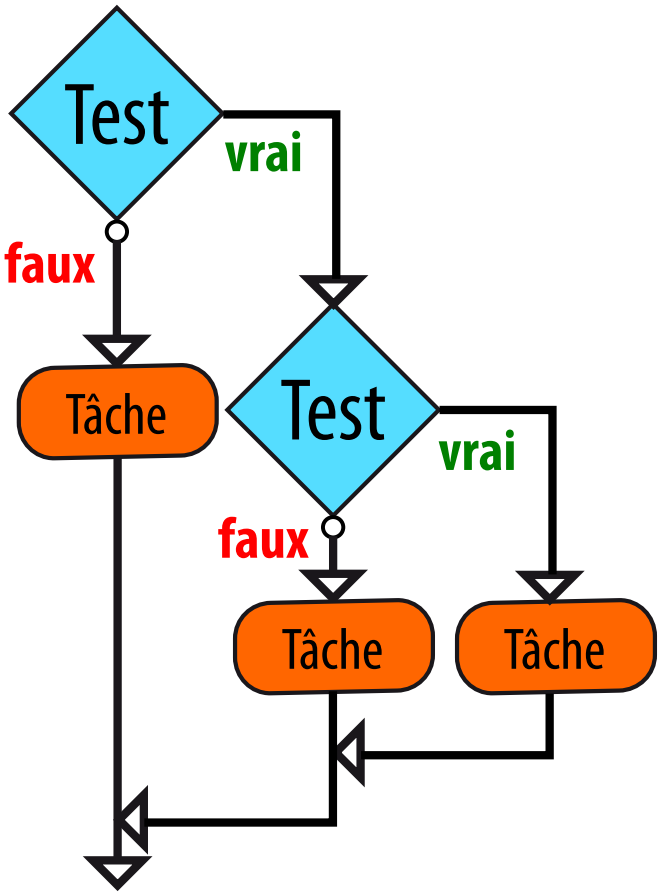
```
if (TEST) {
    // Actions
}

if (TEST) {
    // Actions
} else {
    // Another actions
}
```



Imbrication de blocs { }

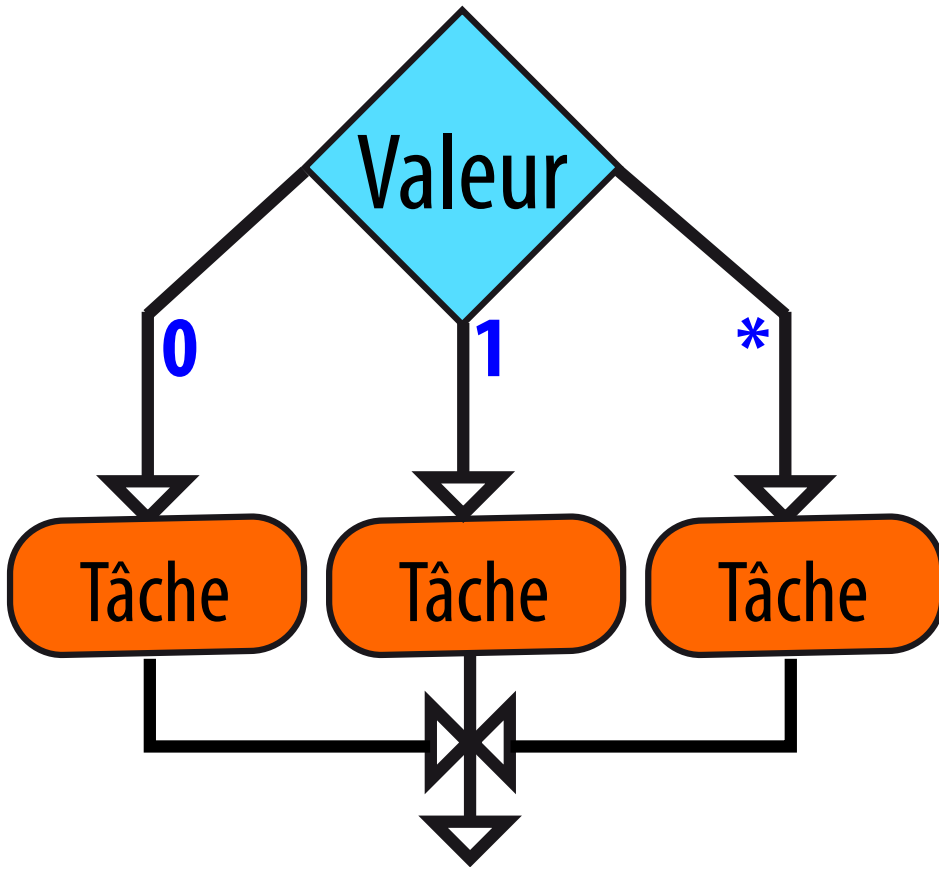
Embranchements

```
if (TEST) {
    if (TEST) {
        // Actions 1
    } else {
        // Actions 2
    }
} else {
    // Actions 3
}
```



Embranchements

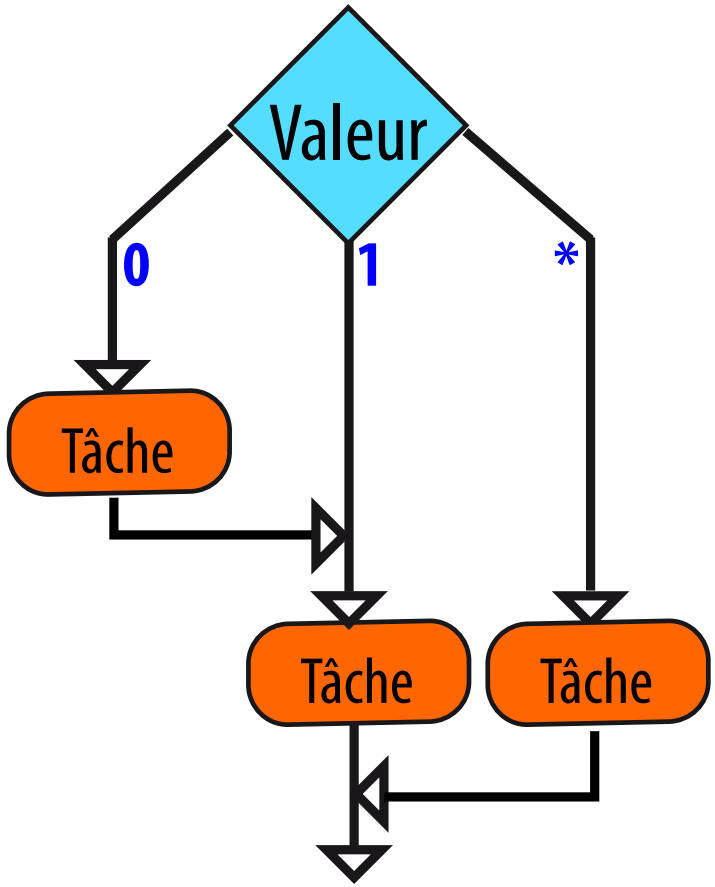
```
switch (VARIABLE) {
    case 0:
        // Actions
        break;
    case 1:
        // Actions
        break;
    default:
        // Actions
        break;
}
```



break pour sortir

Embranchements

```
switch (VARIABLE) {
    case 0:
        // Actions
    case 1:
        // Actions
        break;
    default:
        // Actions
        break;
}
```



Exercices : Qu'affichent ces programmes ?

```c
int main() {
    /* DECLARATION */
    int var1;
    int var2;

    /* EXECUTION */
    var1 = 2;
    var2 = 0;

    if(var1 == 2) {
        // AFFICHE RIRI
    }
    if(var2 > 0) {
        // AFFICHE FIFI
    }
    if(var1 != var2) {
        // AFFICHE LOULOU
    }
    return 0;
}
```

```c
int main() {
    /* DECLARATION */
    int var3;
    int var4;

    /* EXECUTION */
    var3 = 7 / 3;
    var4 = 1;

    if(var3 == 2 && var4 == 1) {
        // AFFICHE BIFUR
    }
    if(var4 <= 1 && var4 > 5 ) {
        // AFFICHE BOFUR
        if(var3 != var4) {
            // AFFICHE BOMBUR
        }
    }
    return 0;
}
```

Exercices : Qu'affichent ces programmes ?

```c
int main() {
    /* DECLARATION */
    int var1;

    /* EXECUTION */
    var1 = 2;

    switch(var1) {
        case 0:
            // AFFICHE PROF
            break;
        case 1:
            // AFFICHE DORMEUR
            break;
        case 2:
            // AFFICHE GRINCHEUX
            break;
    }
    return 0;
}
```

```c
int main() {
    /* DECLARATION */
    int var1;

    /* EXECUTION */
    var1 = 0;

    switch(var1) {
        case 0:
            // AFFICHE PROF

        case 1:
            // AFFICHE DORMEUR
            break;
        case 2:
            // AFFICHE GRINCHEUX
            break;
    }
    return 0;
}
```

Commande pré-processeur

- les lignes commençant par `#` se placent tout en haut du code source
- elles sont appliquées **avant** la compilation
- on retrouve principalement les `#include` et les `#define`
- `#define NOM VALEUR` permet de définir des constantes (à la manière des « Remplacer partout »)

```
#define PI 3.1415926535
```

Avant

```
double rayon;
rayon = 3 * PI;
```

Après

```
double rayon;
rayon = 3 * 3.1415926535;
```