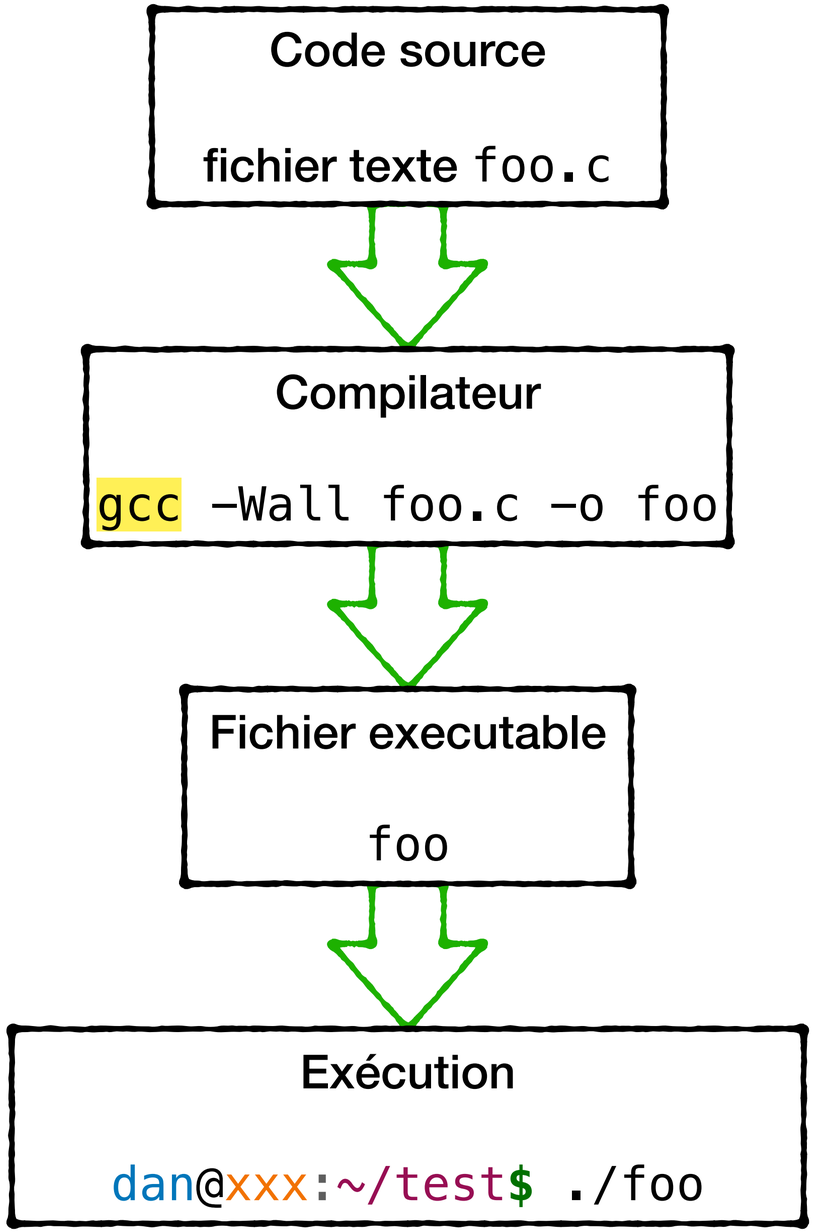
Révision des concepts du C

ICC-C Cours 8

Les épisodes prédédents

- Le langage C est compris par un compilateur qui traduit le code source C en langage machine
- Le compilateur produit un fichier executable qui peut être exécuté par le système d'exploitation
- L'executable est chargé en mémoire et les instructions sont exécutées dans l'ordre par le CPU
- Souvent on utilise un terminal avec un interpréteur de commandes, qui nous permet de facilement lancer d'autres programmes



Code source

- Le code source est composé de déclarations et définitions de fonctions (e.g., `main`)
- Chaque définition de fonction contient une suite d'instructions
- Chaque *programme* a une fonction spéciale dite "d'entrée" (*entry point*) — exécutée quand on lance le programme: `main()`
- Les instructions qui définissent une fonction sont exécutées dans l'ordre

```c
#include <stdio.h>

int main()
{
    printf("Bonjour, ICC!\n");
    return 0;
}
```

Variables, constantes, et types

- On peut réserver des emplacements de mémoire nommés où on peut stocker des valeurs
- Si on compte modifier les valeurs pendant l'exécution, alors ces emplacements s'appellent des variables
- Autrement on les précède de `const` pour obtenir des constantes
- Chaque tel emplacement a un type qui définit
  - la taille en octets qu'occupe l'emplacement en mémoire
  - comment on interprète son contenu

```
int,
char*,
float[5],
void*,
short,
etc.
```

Conversions de type

- Il existe des types de base: `char`, `int`, `long`, `float`, `double`
- Pour chaque type `T` (de base ou non) on peut définir
  - un type tableau d'éléments de ce type: `T[]`
  - un type pointeur vers ce type: `T*`
- On peut convertir explicitement un type vers un autre
  `(float) 3 -> 3.0`
  - Parfois ça n'a pas de sens… `(float) &variable`

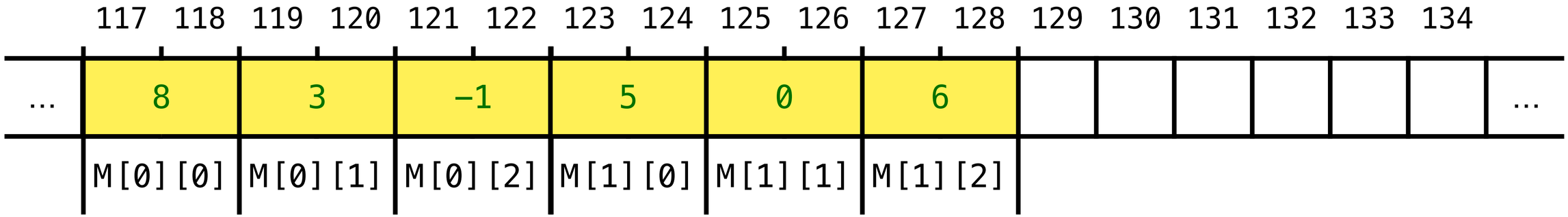
Emplacements de mémoire

- Les variables et constantes sont souvent locales
  - Vivent sur la pile d'exécution (*stack*) — allocation statique
  - Valides uniquement pendant l'exécution de la fonction où elles sont définies
- On peut définir des variables globales
  - Accessibles pour toutes les fonctions — dangereux! ⛔
- On peut demander des emplacements de mémoire persistants sur le tas (*heap*) — allocation dynamique
  - `malloc` pour obtenir un pointeur vers l'emplacement demandé; `free` pour les libérer

Tableaux

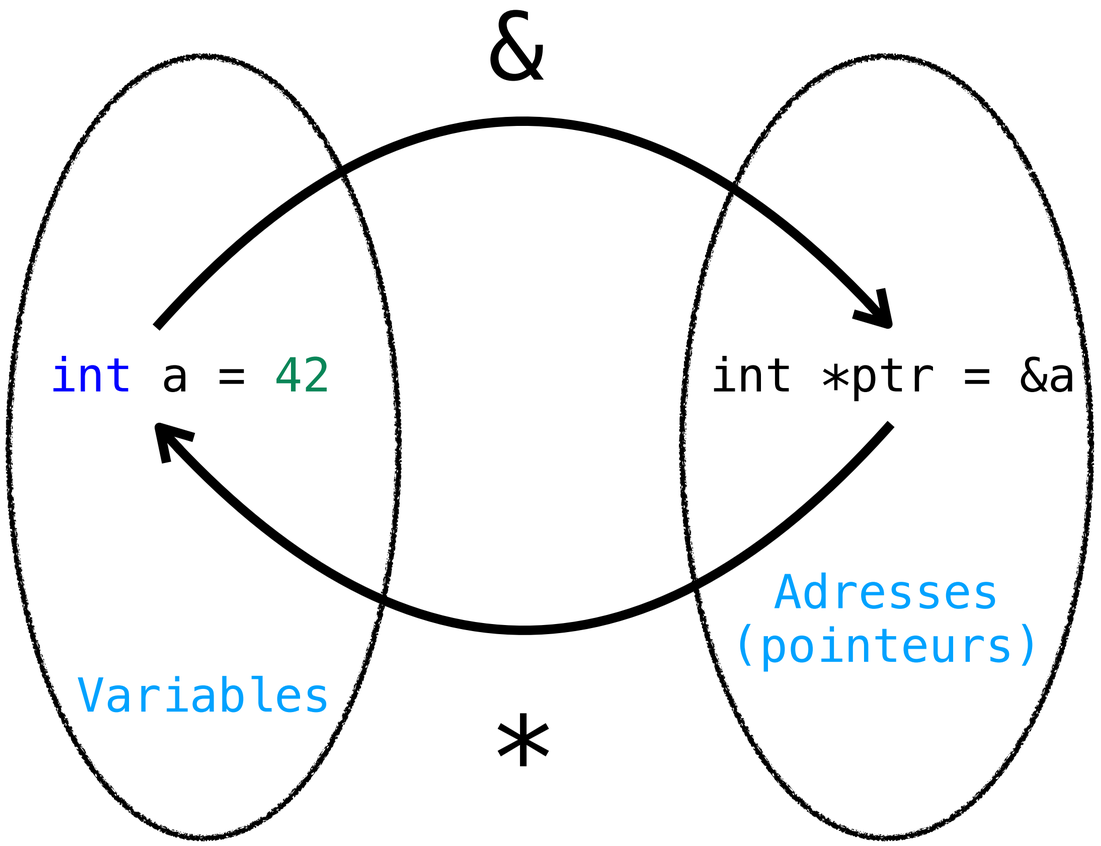
- Un tableau d'éléments d'un certain type `T` a le type `T[]`
- Il a une taille fixe de k éléments
- Il occupe une zone contiguë de mémoire de `k x sizeof(T)`
- On peut définir un **tableau de tableaux**
  - C'est aussi une zone contiguë de mémoire!

```
short M[2][3] = {
    {8, 3, -1},
    {5, 0, 6},
};
```



Pointeurs

- Un pointeur vers un emplacement d'un certain type T a le type T*
- On obtient un pointeur vers une variable avec l'opérateur &
- Pour utiliser l'emplacement où pointe un pointeur, on utilise l'opérateur *



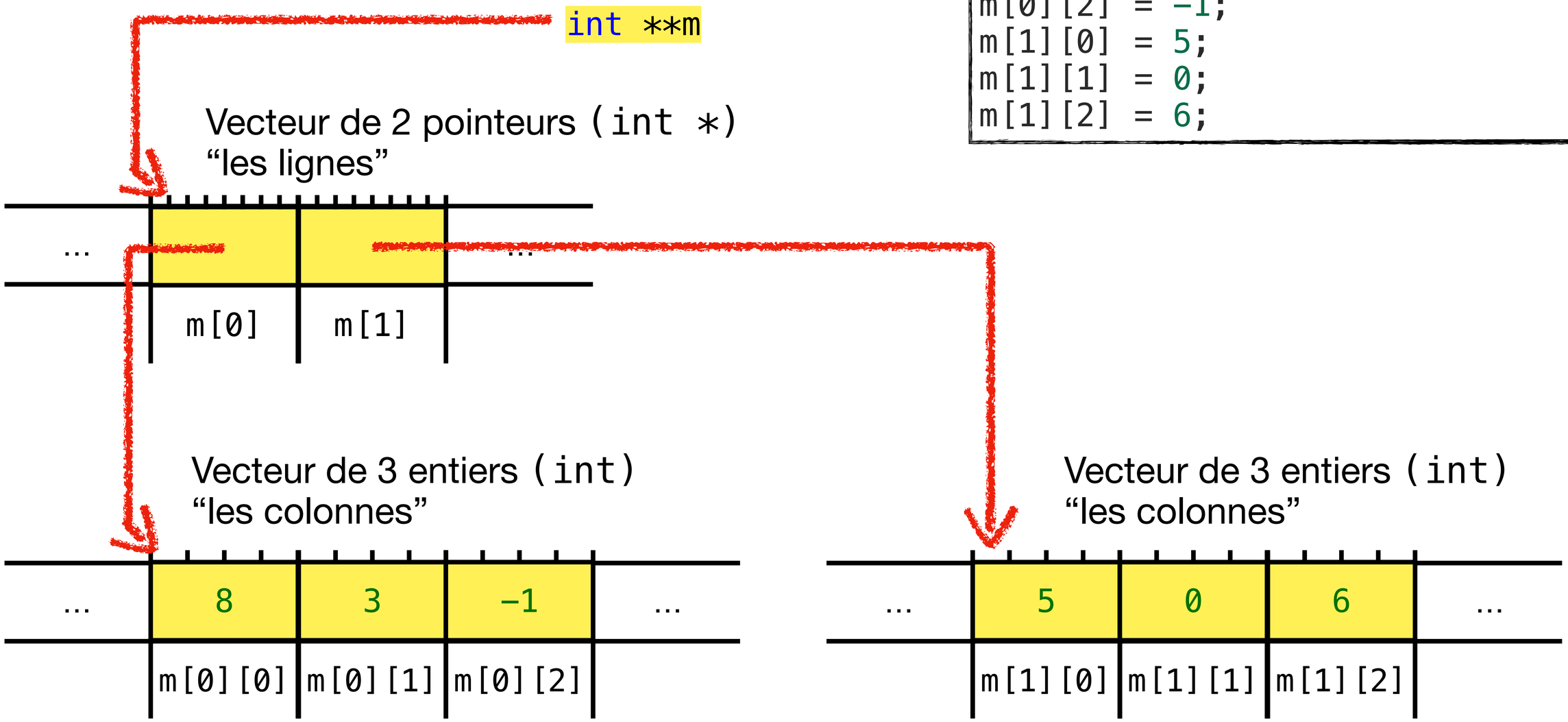
Pointeurs doubles

- On peut même définir un pointeur vers une variable de type pointeur
- le type “pointeur vers T*” est T**
- Allocation dynamique d’une matrice

```
int **creer_matrice(int m, int n)
{
    int **mat = malloc(m * sizeof(int*));
    // mat est maintenant un tableau
    // de m pointeurs vers int
    for (int i=0; i<m; i++)
    {
        mat[i] = malloc(n * sizeof(int));
        // mat[i] pointe vers un tableau
        // de n entiers
    }
    return mat;
}
```

Pointeurs doubles

```c
int **m = creer_matrice(2, 3);
m[0][0] = 8;
m[0][1] = 3;
m[0][2] = -1;
m[1][0] = 5;
m[1][1] = 0;
m[1][2] = 6;
```



Expressions et opérateurs

- Une expression est formée d'un enchaînement d'opérations sur des valeurs
- Elle a un type et peut être évaluée pour obtenir une valeur à l'exécution
- Il existe des opérations unaires, binaires, et même ternaires
- L'ordre des opérations est important, ainsi que l'associativité (g-à-d, ou d-à-g)
- Les opérations booléennes &&, ||, !, ==, !=, >, <, etc., utilisent la règle "0 est faux, tout autre valeur est vraie"
- Afin de faciliter l'écriture il y a des conversions implicites de type

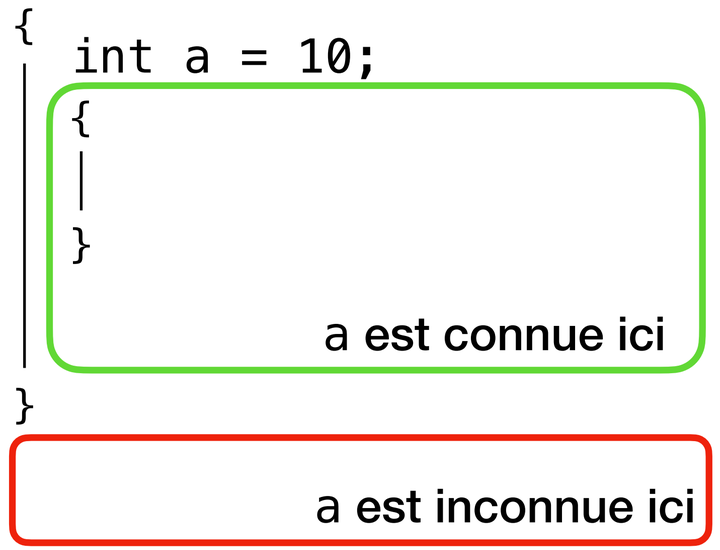
Expressions et opérateurs

- Certaines opérations ont des effets secondaires sur les opérandes
  - Dans ce cas les opérandes doivent être des L-valeurs
  - Exemples: affectation (=, +=, /=, ...), incrémentation (++, −−)
- 🙏 Dans le doute, utilisez les parenthèses!
- ⚠️ Confusion possible entre affectation (=) et test d'égalité (==)
- ⚠️ Confusion possible entre et, ou (&&, ||) et et-par-bits, ou-par-bits (&, |)

Instructions & contrôle de flux

Flow control

- Une instruction simple est une expression suivie d'un point-virgule `;`
- Une déclaration de variables/constantes constitue aussi une instruction
- Une instruction composée / bloc est une suite d'instructions (y compris des blocs!) entourées d'accolades
- ⚠️ Attention à la portée des variables



Instructions & contrôle de flux

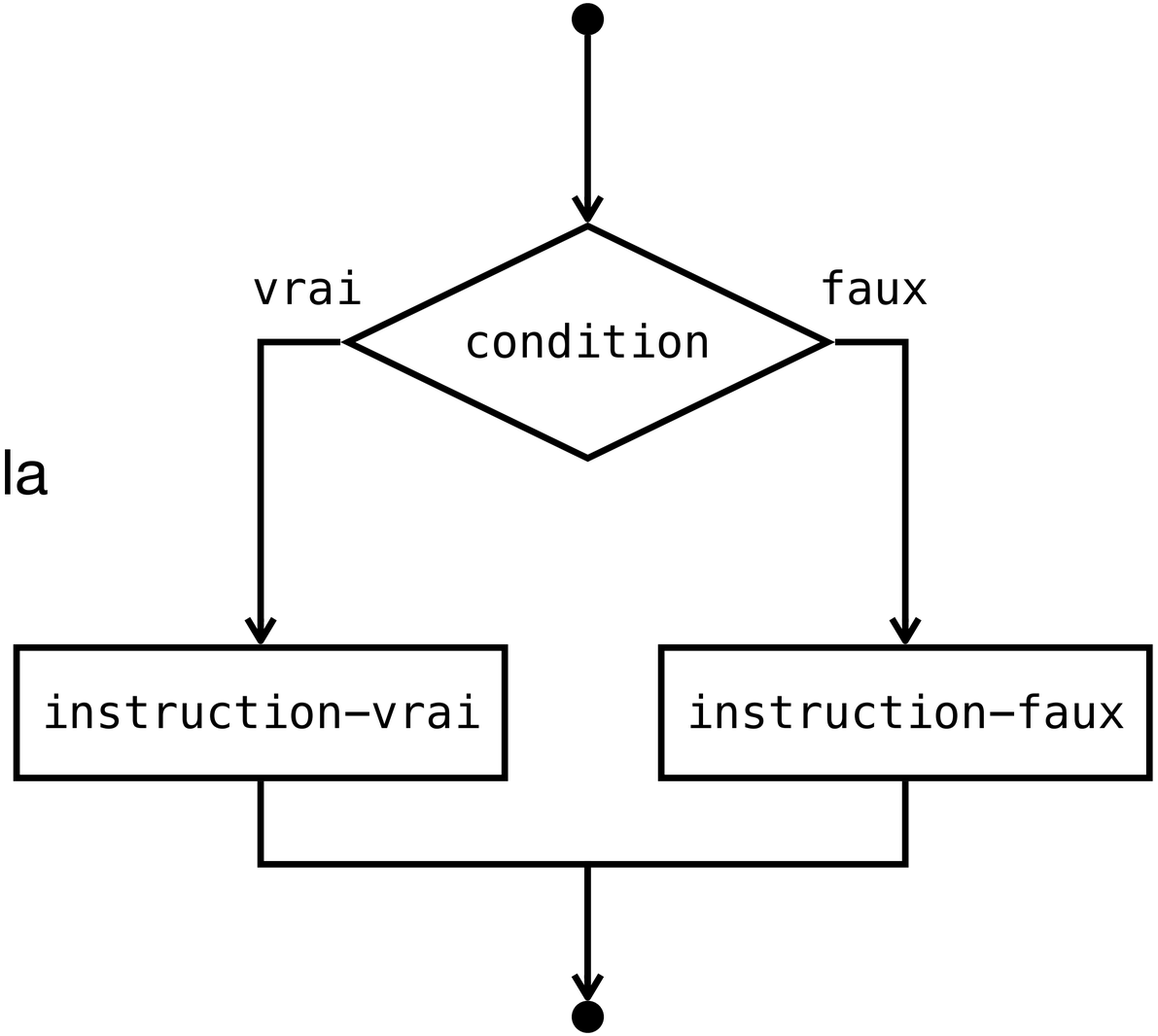
`if`

- L’instruction conditionnelle if décide quel code exécuter selon une condition

```
if (<condition>) instruction-vrai
```

- On peut aussi spécifier l’alternative avec la clause “else”:

```
if (<condition>) instruction-vrai
else instruction-faux
```

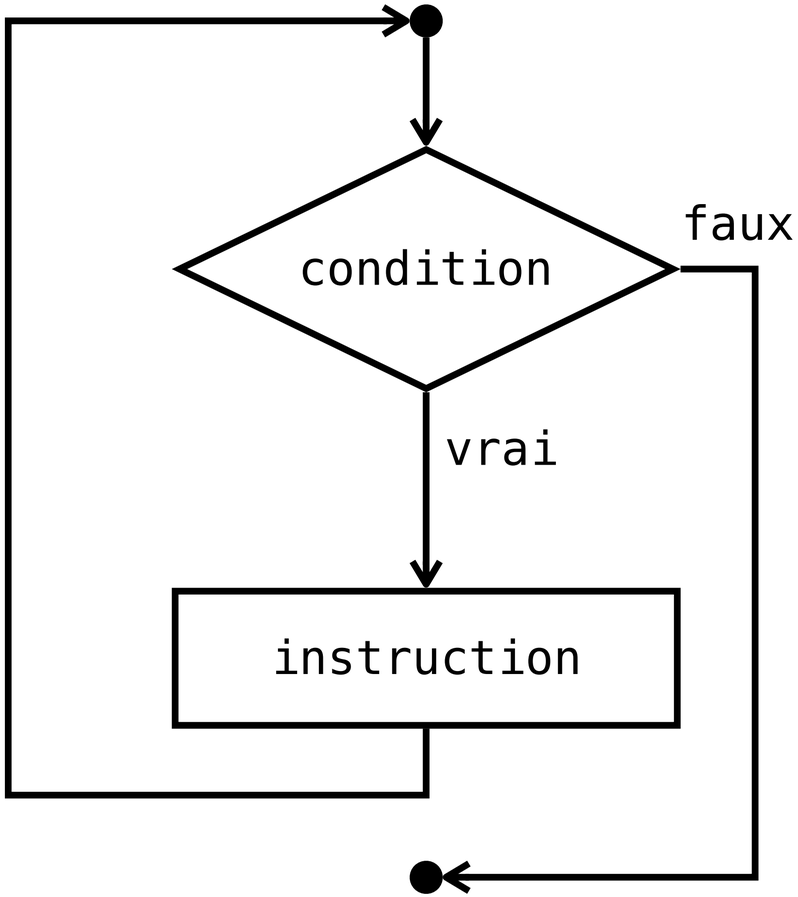


Instructions & contrôle de flux
`while`

- Une boucle `while` a la syntaxe suivante:

```
while (condition) instruction
```

Tant que `condition` est vraie execute l'`instruction`.
Dès que `condition` n'est plus vraie, continue à l'instruction d'après.



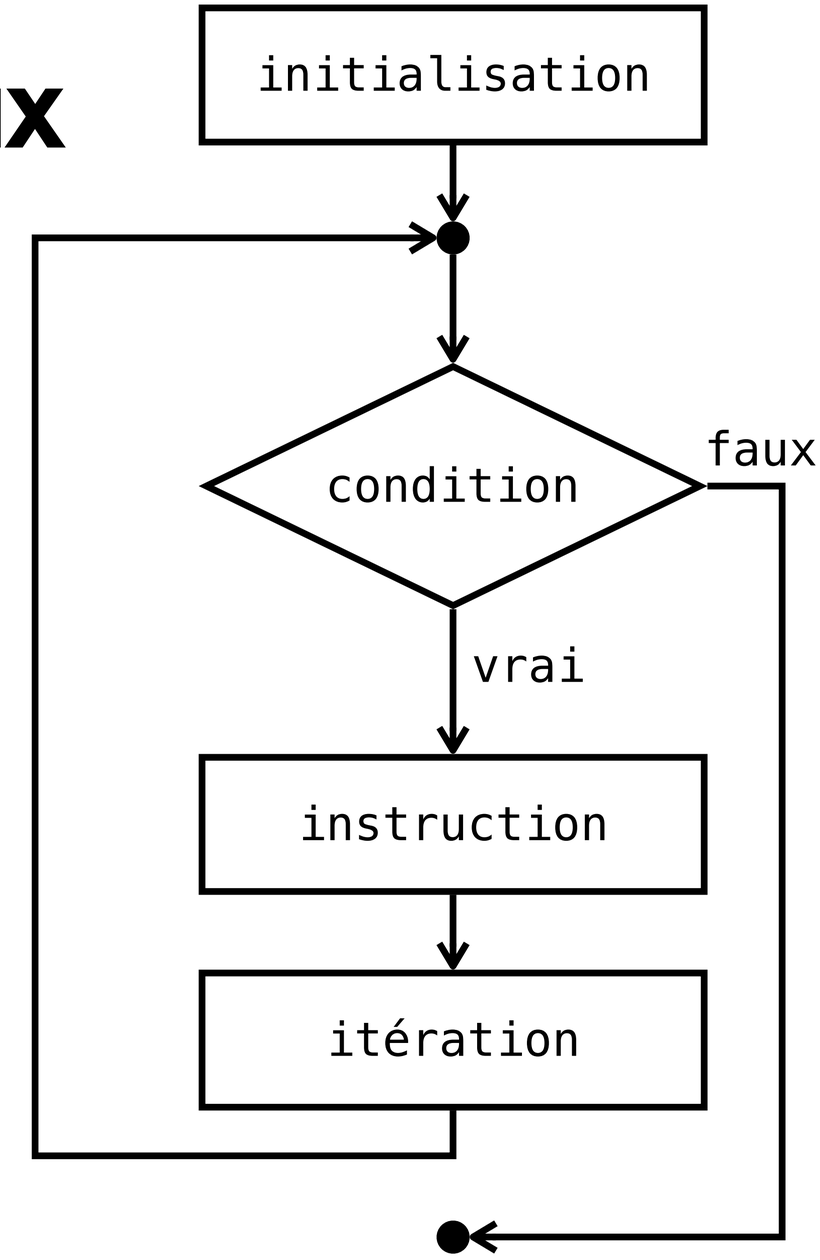
Instructions & contrôle de flux

`for`

```
for (initialisation; condition; itération)
instruction
```

- Est équivalente à:

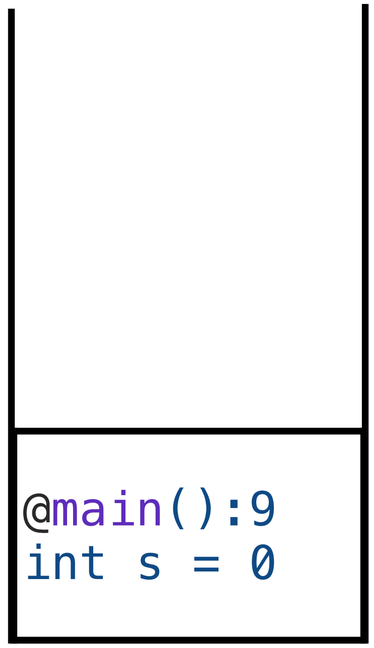
```
{
  initialisation;
  while (condition)
  {
    instruction;
    itération;
  }
}
```



La pile d’exécution

Call stack

- Le contexte des fonctions est stocké dans la pile d’exécution



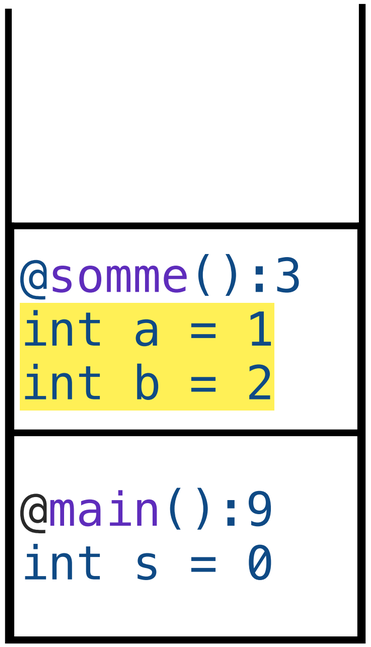

```c
int somme(int a, int b)
{
  return a + b;
}

int main()
{
  int s = 0;
  s = somme(1, 2); // Arguments 1 et 2
  printf("%d\n", s);
  // Affiche: 3
}
```

La pile d’exécution

Call stack

- Quand on appelle une fonction, on `push` son nouveau contexte dans la pile
- Les valeurs des arguments sont **copiées** dans les paramètres
  — passage par valeur



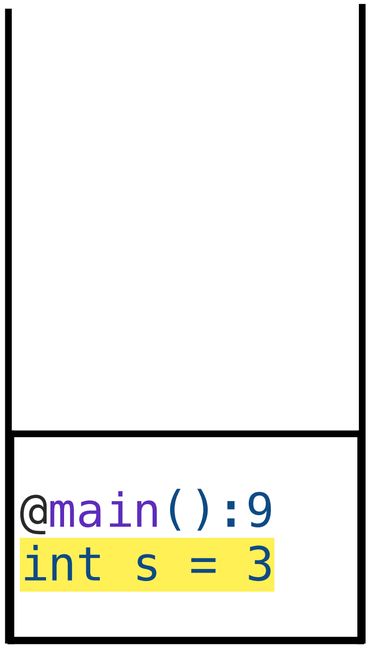

```c
int somme(int a, int b)
{
  return a + b;
}

int main()
{
  int s = 0;
  s = somme(1, 2); // Arguments 1 et 2
  printf("%d\n", s);
  // Affiche: 3
}
```

La pile d’exécution

Call stack

- `return valeur` = on quitte la fonction & `pop` son contexte de la pile
- La valeur se retrouve dans l’expression où l’appel a été effectué




```c
int somme(int a, int b)
{
  return a + b;
}

int main()
{
  int s = 0;
  s = somme(1, 2); // Arguments 1 et 2
  printf("%d\n", s);
  // Affiche: 3
}
```

Debugging 🪲

“Déboggage” 🤢

- Il y a un problème dans mon code
- Que fais-je?
  - Option 1: Je demande à un assistant
  - Option 2: Je demande à ChatGPT
  - Option 3: Je mets des `printf` partout

Debugging 

- Les `printf` peuvent nous aider à comprendre ce qui ne va pas!
- C’est bien de les utiliser avec parcimonie pour éviter les confusions
- On veut suivre le cheminement dans le code qui a pu mener au comportement inattendu
- On affiche aux endroits clé

Debugging 🪲

- Parfois on peut utiliser un outil qui s'appelle **le Debugger**
- Le debugger original = `gdb` - pas d'interface graphique 😬
- Concepts importants:
  - Breakpoint = point où l'execution s'arrêtera pour inspection de l'état 

  -  Step-in = rentrer dans l'execution d'une fonction
  -  Step-out = executer jusqu'à sortir d'une fonction
  -  Step-over = évaluer l'instruction suivante, sans suivre les appels de fonction

Demo

Projet!