Les fonctions

thibaut.lust@lip6.fr

Polytech Sorbonne

2020

`https://moodle-sciences.upmc.fr` (cours Informatique Générale EPU-R5-IGE)

Cours basé sur les diapositives créées par Julien Brajard

Plan du cours

Introduction

- La résolution d'un problème informatique peut conduire à la résolution de plusieurs problèmes plus élémentaires.
  - Exemple des traitements des tableaux
- Idée :
  - Identifier ces traitements élémentaires pour la résolution du problème initial.
  - Construire des petits programmes pour chacun des traitements élémentaires (et les tester).
  - Ecrire un programme final simple qui utilise les programmes élémentaires comme des briques de base.

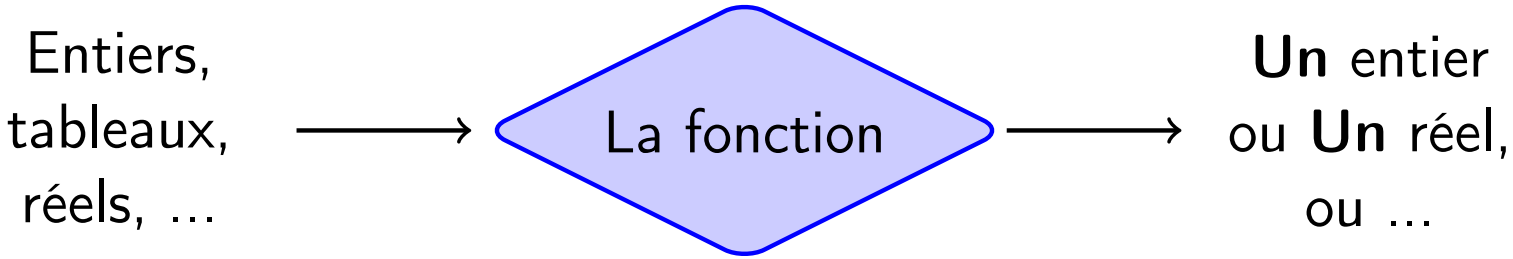
Solution : les fonctions

Le langage C permet ce découpage. C'est la programmation modulaire.

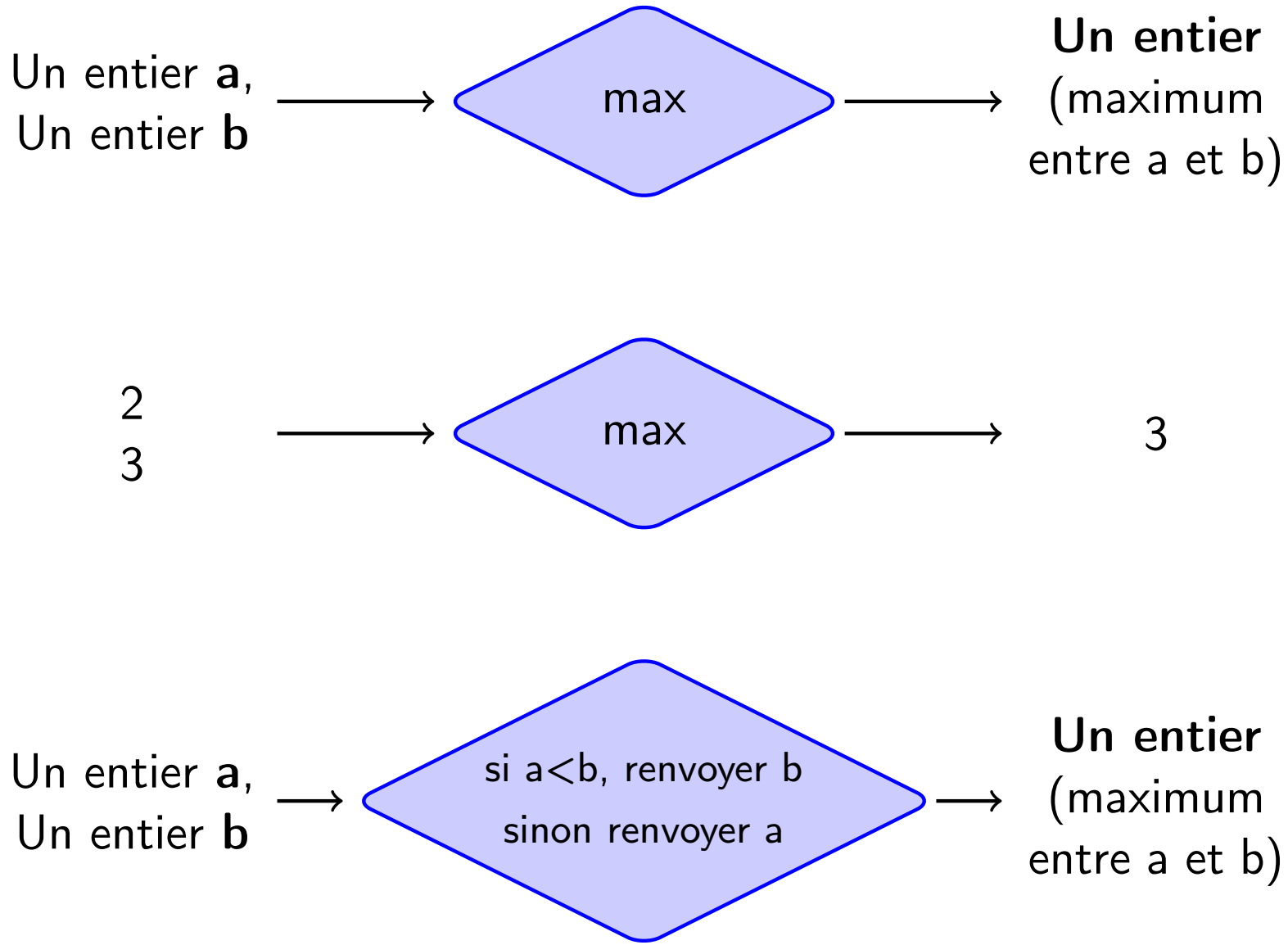
- Permet de découper un traitement en autant de traitements élémentaires qu'on le souhaite.
- Permet d'appeler des modules existants (bibliothèque).
- Permet de définir ses modules et de construire ses propres bibliothèques.
- Permet de découper un gros logiciel pour faciliter sa mise au point.
- Evite des copier-coller de code et améliore sa visibilité.
- Permet le travail collaboratif.

Qu'est-ce qu'une fonction ?

Une fonction est un sous-programme pouvant être appelé dans un programme et qui effectue une suite d'opérations.
Les opérations effectuées peuvent dépendre de 1 ou plusieurs entrées et la fonction peut renvoyer au plus une seule sortie (en langage C).

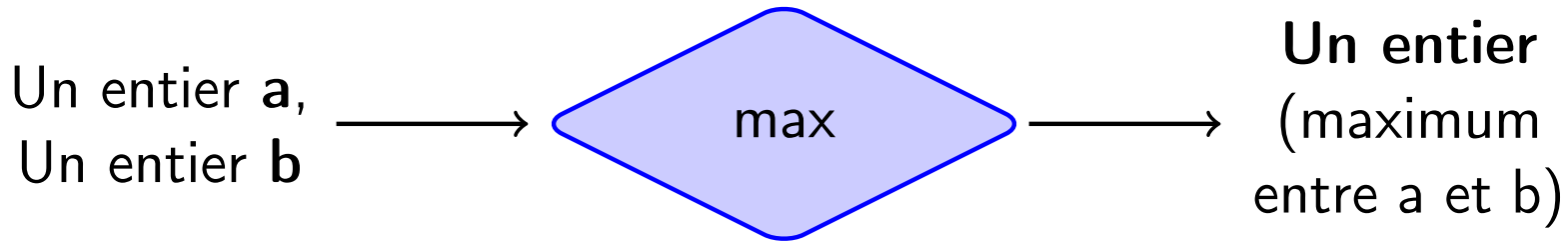


Exemple : la fonction max

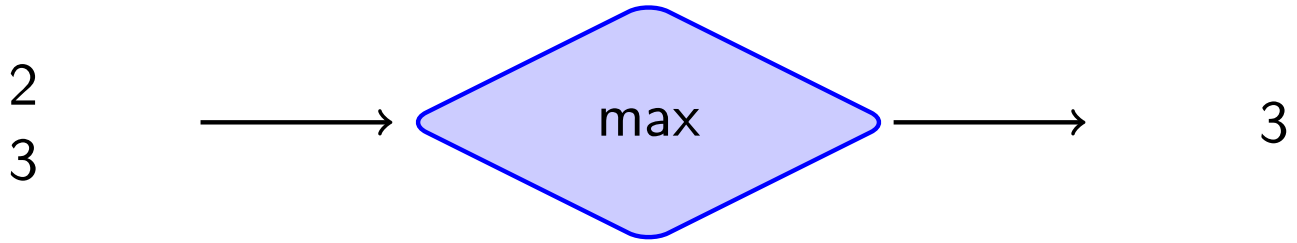


Terminologie

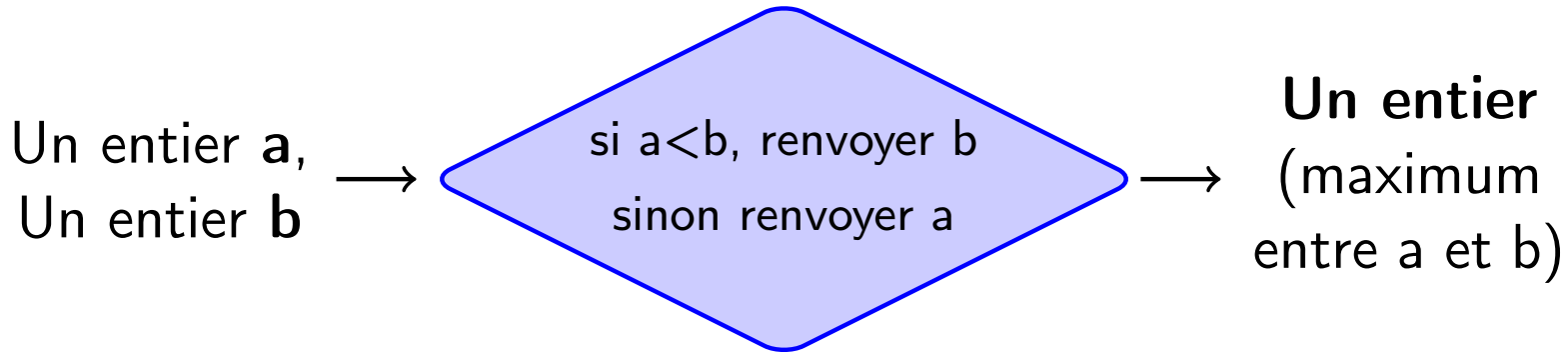
Déclaration :



Appel :



Définition :



En langage C

- Déclaration (le prototype de la fonction) :
  ```
  type_retour nom_fonction(arguments);
  ```
- Appel :
  ```
  nom_fonction(liste_valeurs);
  ```
- Définition :
  ```
  type_retour nom_fonction(arguments)
  {
    déclaration de variables locales
    bloc d'instructions
    return valeur_retour;
  }
  ```

La fonction max

- Déclaration :

```
int max (int a, int b);
```

- Appel :

```
int maxi ;
int n ;
scanf("%d",&n);
maxi = max(n,3); //Appel de la fonction max
```

- Définition :

```
int max (int a, int b)
{
  if (a<b) return(b) ;
  else return(a) ;
}
```

Quelque règles

- Une fonction peut être définie n'importe où dans le programme ;
- Une fonction doit être déclarée avant d'être appelée pour la première fois ;
- Une fonction peut être appelée dans le `main` ou dans une autre fonction ;
- On ne peut pas définir une fonction dans une autre fonction ;
- Une fonction peut prendre autant d'arguments que l'on souhaite en entrée mais ne retourne qu'un plus un élément ;
- Les règles sur les noms de fonctions sont les mêmes que sur les noms de variables.

Tolérance

Si l'on définit une fonction avant de l'appeler, on peut éventuellement se passer de la déclarer.

```c
//Déclaration-Définition
int max (int a, int b) {
  if (a<b) return (b);
  else return (a);
}

//Fonction principale
int main() {
  int maxi, n ;
  scanf("%d",&n);
  maxi = max (n,3);
  printf("%d\n",maxi);
  return -1;
}
```

Equivalent

$\Leftrightarrow$

```c
//Déclaration
int max (int a, int b) ;

//Fonction principale
int main() {
  int maxi, n ;
  scanf("%d",&n);
  maxi = max (n,3);
  printf("%d\n",maxi);
  return -1;
}

//Définition
int max (int a, int b) {
  if (a<b) return (b);
  else return (a);
}
```

Fonction appelée / Fonction appelante

```c
//Déclaration des fonctions
int f(int a);
int g(int b);

//Fonction principale
int main() {
  int z = 2;
  z = g(z) + f(z);
  printf("%d\n",z);
  return -1;
}

//Définition des fonctions
int f (int a) {
  return (a+2);
}

int g (int b) {
  return (2*f(b));
}
```

Une fonction peut être appelée de n'importe quelle fonction y compris dans le `main`

- Fonctions appelantes :
  - `main` appelle `g` et `f`
  - `g` appelle `f`
- Fonctions appelées :
  - `f` est appelée par `main` et `g`
  - `g` est appelée par `main`

Entrées et sorties

Prototype d'une fonction

```
type_retour nom_fonction(arguments);
```

- Les entrées de la fonction sont appelées arguments.
- *type_retour* est le type renvoyé par la fonction. Une fonction (en C) ne peut renvoyer qu'une valeur de ce type.

Les sorties

L'instruction `return`

Une fonction retourne une valeur à l'appellant par l'instruction `return`.

- Syntaxe : `return(`*expression*`);`
- Exemples :

```
return (-1);
```

```
return (2*z + 3);
```

```
return (2 * f(1+z));
```

- La valeur de retour est convertie selon le type de la valeur de retour précisé dans le prototype de la fonction.
- Parfois, le compilateur émet un avertissement (`Warning`) si ce n'est pas le cas.
- La fonction appelante n'utilise pas forcément la valeur de retour.

Les sorties

La sortie `void`

- Une fonction qui ne retourne rien à l'appelant doit avoir pour type de retour `void`.
- Exemple : Les fonctions faisant des affichages.

```
void affiche_intervalle(int a, int b)
{
  int j;
  for (j=a;j<=b;j++)
  {
    printf("%d\t",j);
  }
  printf("\n");
}
```

Les entrées

Argument formel / Argument effectif

Argument formel

Nomme et décrit le type des arguments à l'intérieur de la fonction

- Utilisé lors de la déclaration ou de la définition de la fonction.

Argument effectif

Valeur qui sera affectée à l'argument formel.

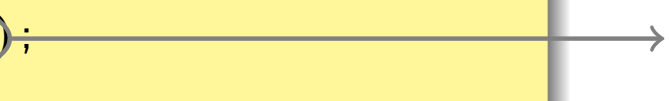
- Utilisé lors de l'appel de la fonction.

Exemple 1

```c
// Déclaration de la fonction
int max (int a, int b);

//Programme principal
int main() {
  int maxi;
  int n;
  scanf("%d",&n);
  maxi = max(n, 3);
  return -1;
}

//Définition de la fonction
int max (int a, int b) {
  if (a<b) return(b);
  else return(a);
}
```

Exemple 1

```c
// Déclaration de la fonction
int max (int a, int b);

//Programme principal
int main() {
  int maxi;
  int n;
  scanf("%d",&n);
  maxi = max(n, 3);
  return -1;
}

//Définition de la fonction
int max (int a, int b) {
  if (a<b) return(b);
  else return(a);
}
```



argument formel

argument effectif

argument formel

Exemple 2

```
// Déclaration de la fonction
float moins (float x, float y);

//Programme principal
int main() {
  float x,y,z1,z2,z3,z4;
  x = 2; y = 3;
  z1 = moins (6,y);
  z2 = moins (6,x);
  z3 = moins (x,y);
  z4 = moins (y,x);
  return -1;
}

//Définition de la fonction
float moins(float x, float y) {
  return (x-y) ;
}
```

- z1=
- z2=
- z3=
- z4=

La conversion automatique des arguments

```c
// Déclaration de la fonction
float moitie (int n);

//Programme principal
int main() {
   float x = 3.5;
   x = moitie(x);
   printf("\nx=%f\n",x);
   return -1;
}

//Définition de la fonction
float moitie(int n) {
   float x;
   x = (float)n/2;
   return(x);
}
```

Qu'affiche le programme ?

Le contexte d'exécution

Chaque fonction a son contexte d'exécution propre :

- Les variables déclarées dans une fonction ou déclarées en paramètres ne sont connues qu'à l'intérieur de cette fonction.
- Toute modification de ces variables dans la fonction n'a pas d'impact pour les variables déclarées ailleurs.

Exemple sans fonction

Contexte d'exécution du `main`

```
int main() {
  int a ;
```

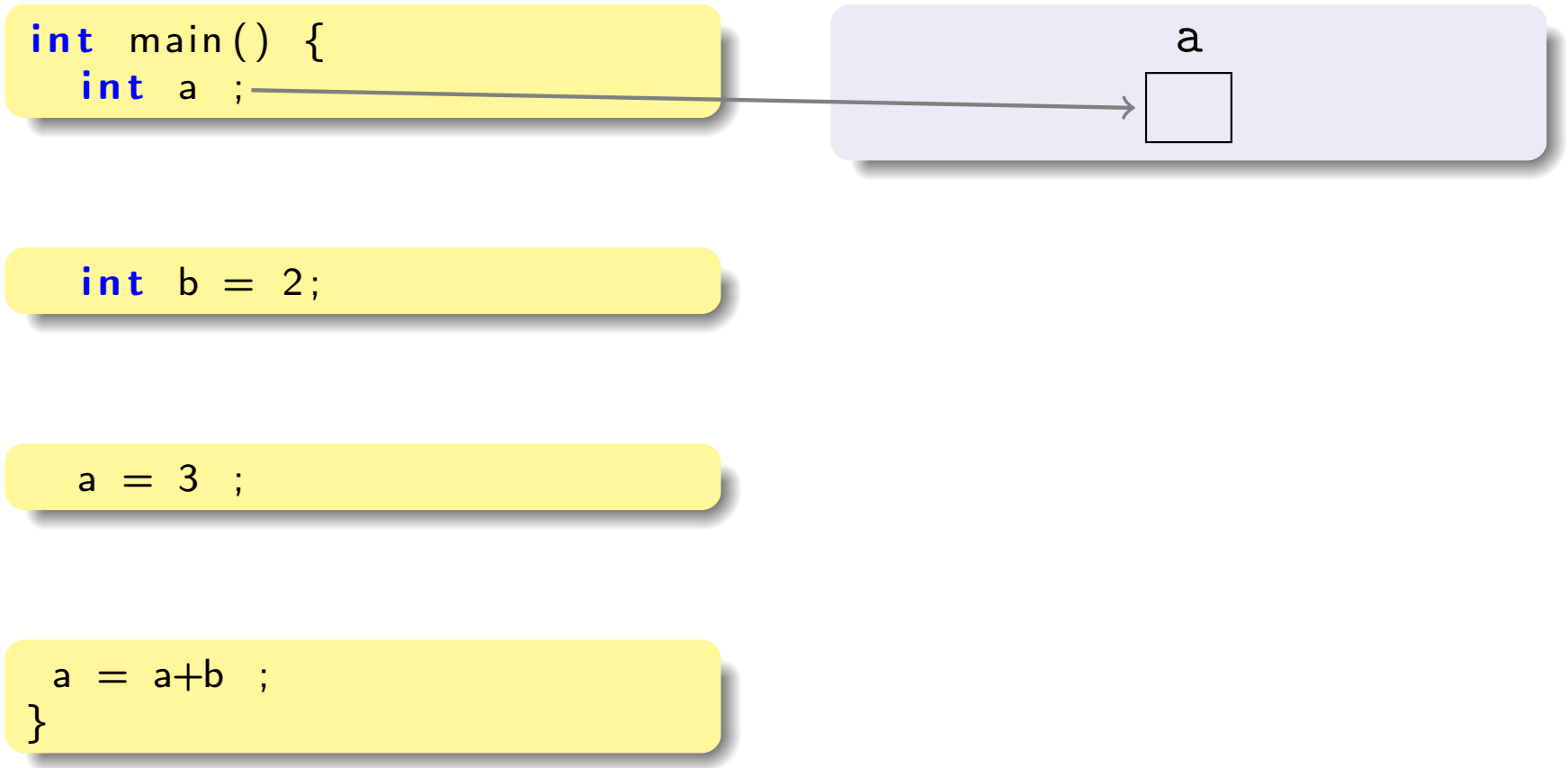
```
  int b = 2;
```

```
  a = 3 ;
```

```
 a = a+b ;
}
```

Exemple sans fonction

Contexte d'exécution du `main`



Exemple sans fonction

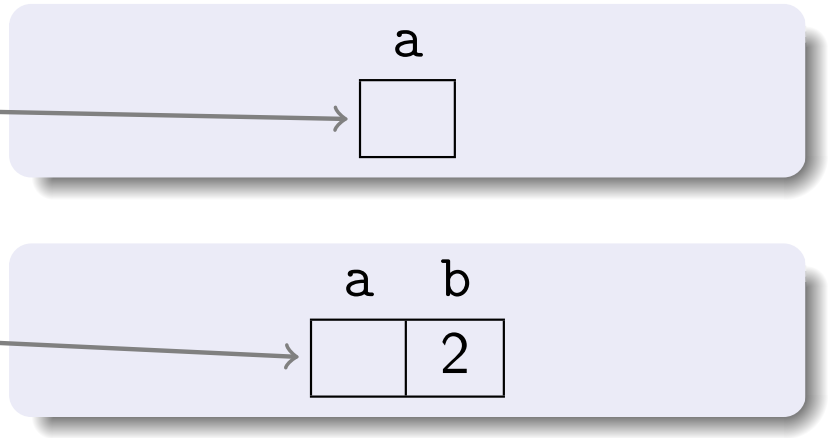
```
int main() {
  int a ;
```

```
  int b = 2;
```

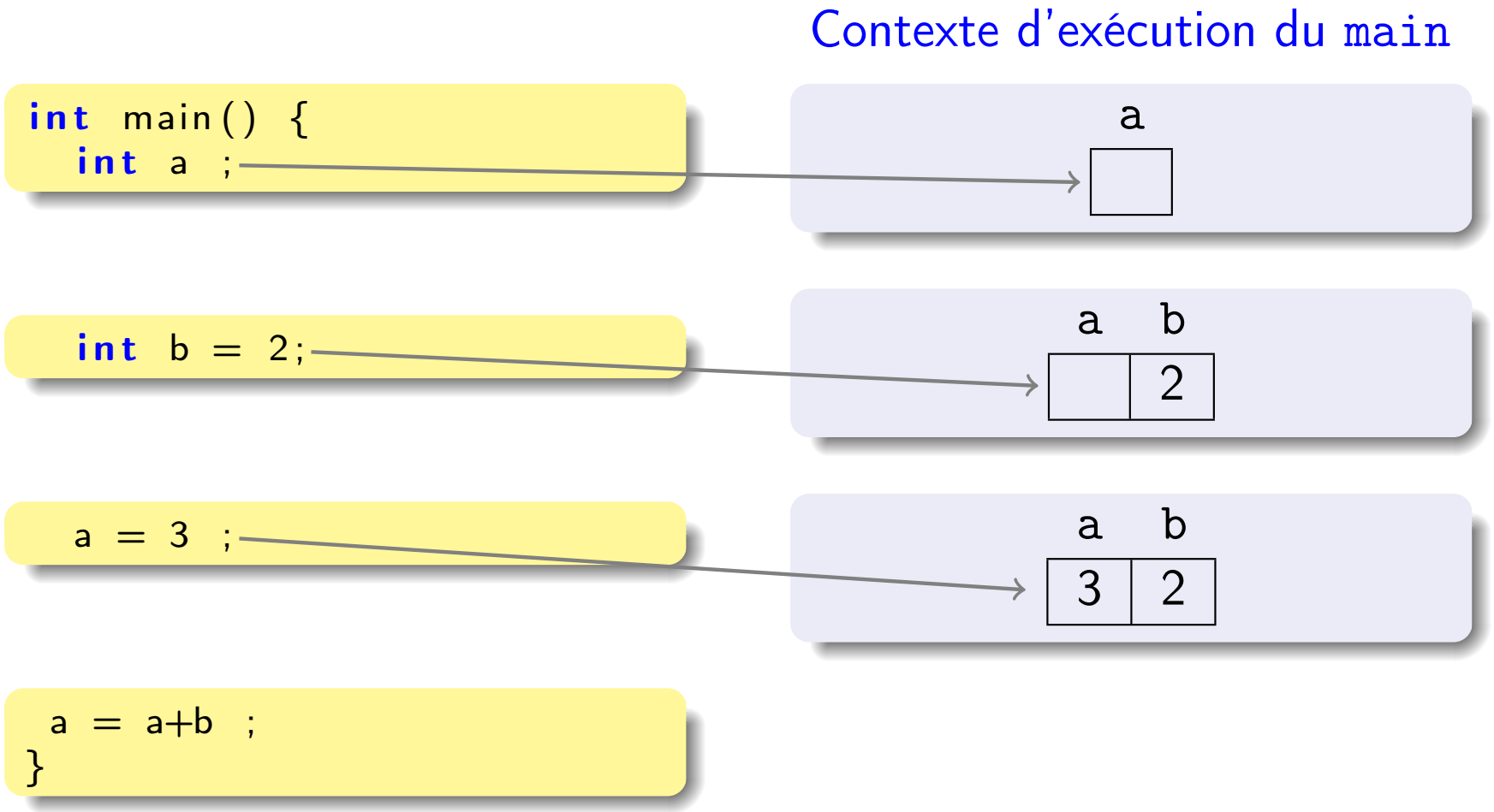
```
  a = 3 ;
```

```
  a = a+b ;
}
```

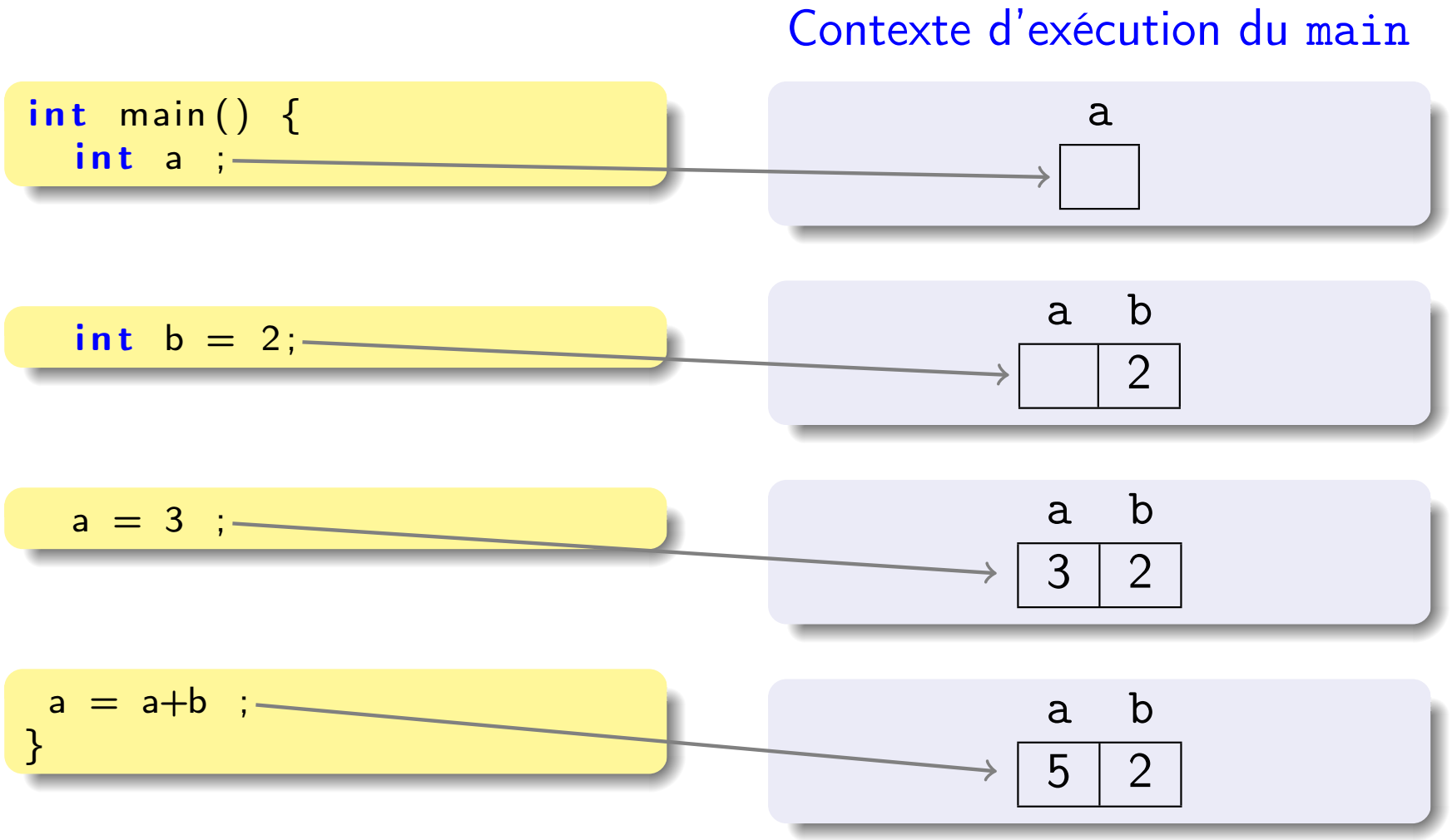
Contexte d'exécution du `main`



Exemple sans fonction



Exemple sans fonction



Exemple avec fonction

```
int main() {
  int x=2, q=3, p;
  p=puiss(x,q);
  return -1;
}

int puiss(int x, int n) {
  int y = 1;
  while (n>0){
    y=y*x ;
    n--;
  }
  return(y);
}
```

Exemple avec fonction

Contexte du `main`

```
int main() {
 int x=2,q=3,p;
```

x	q	p
2	3	

```
 p=puiss(x,q);
```

```
}
```

Exemple avec fonction

Contexte du `main`

Contexte de `puiss`

```
int main() {
 int x=2,q=3,p;
```

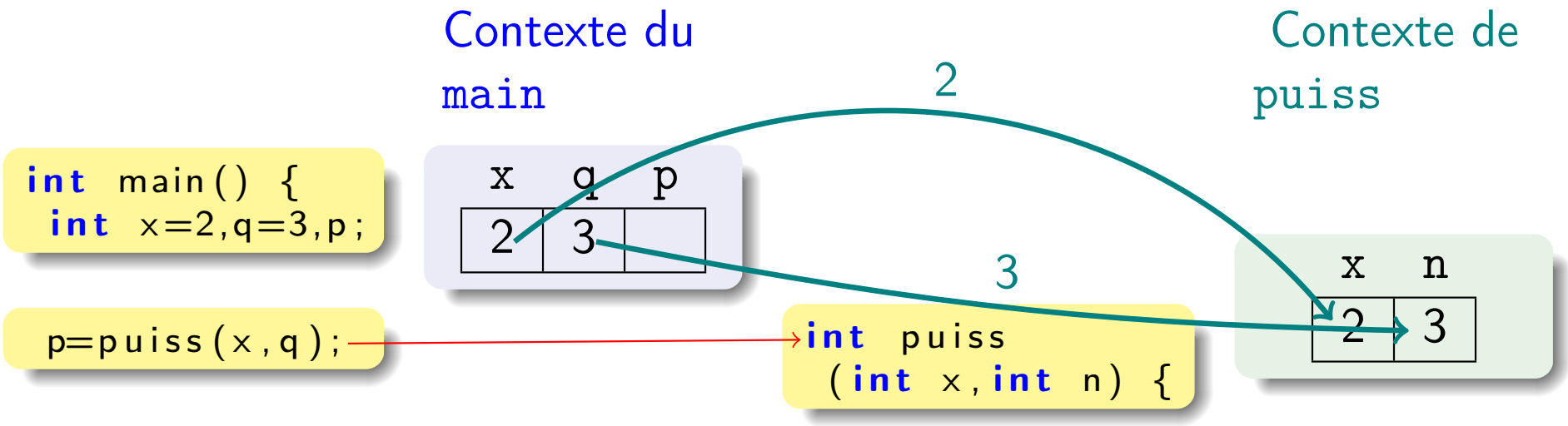
x	q	p
2	3	

```
 p=puiss(x,q);
```

```
int puiss
 (int x,int n) {
```

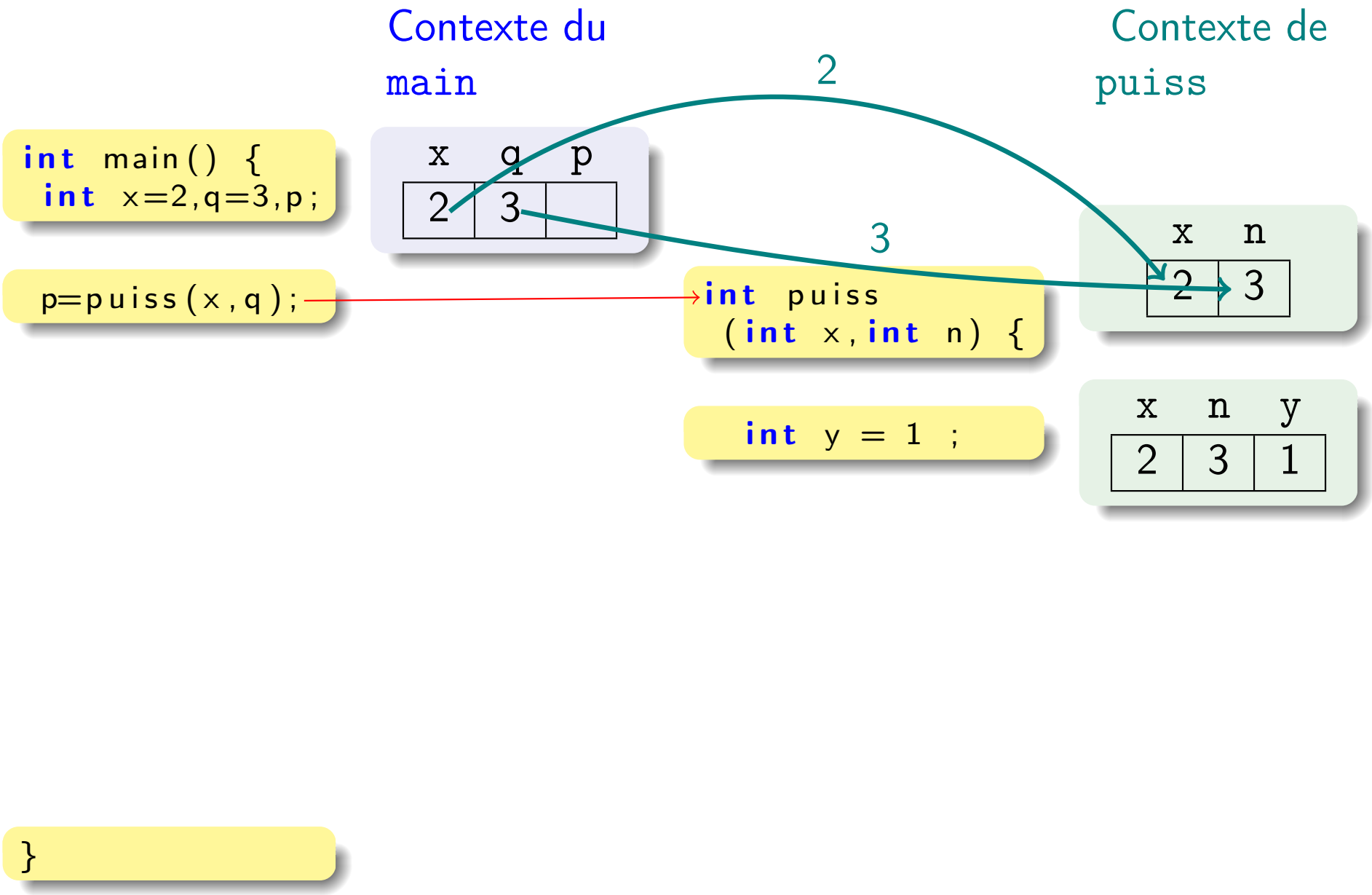
```
}
```

Exemple avec fonction

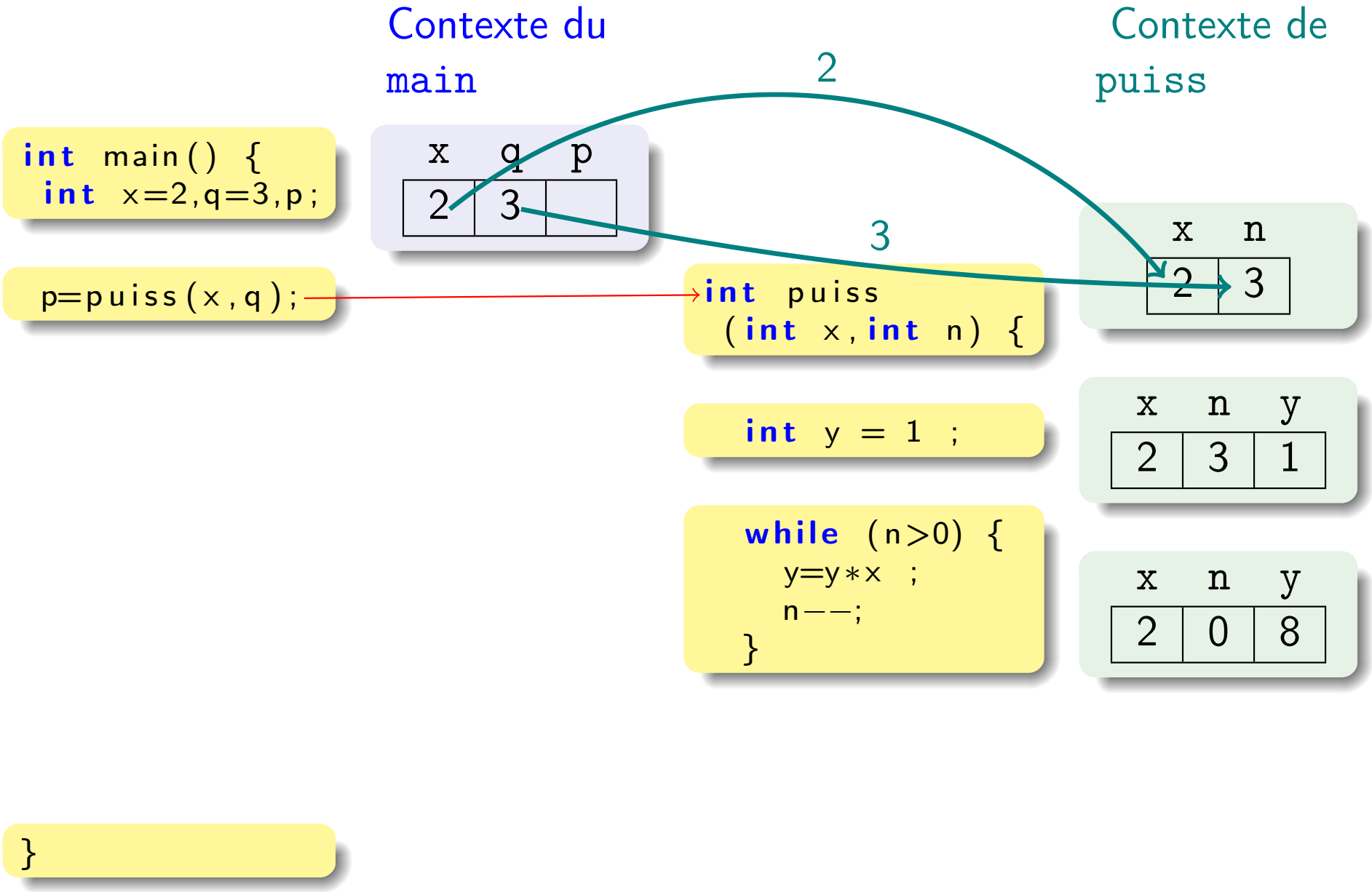



```
}
```

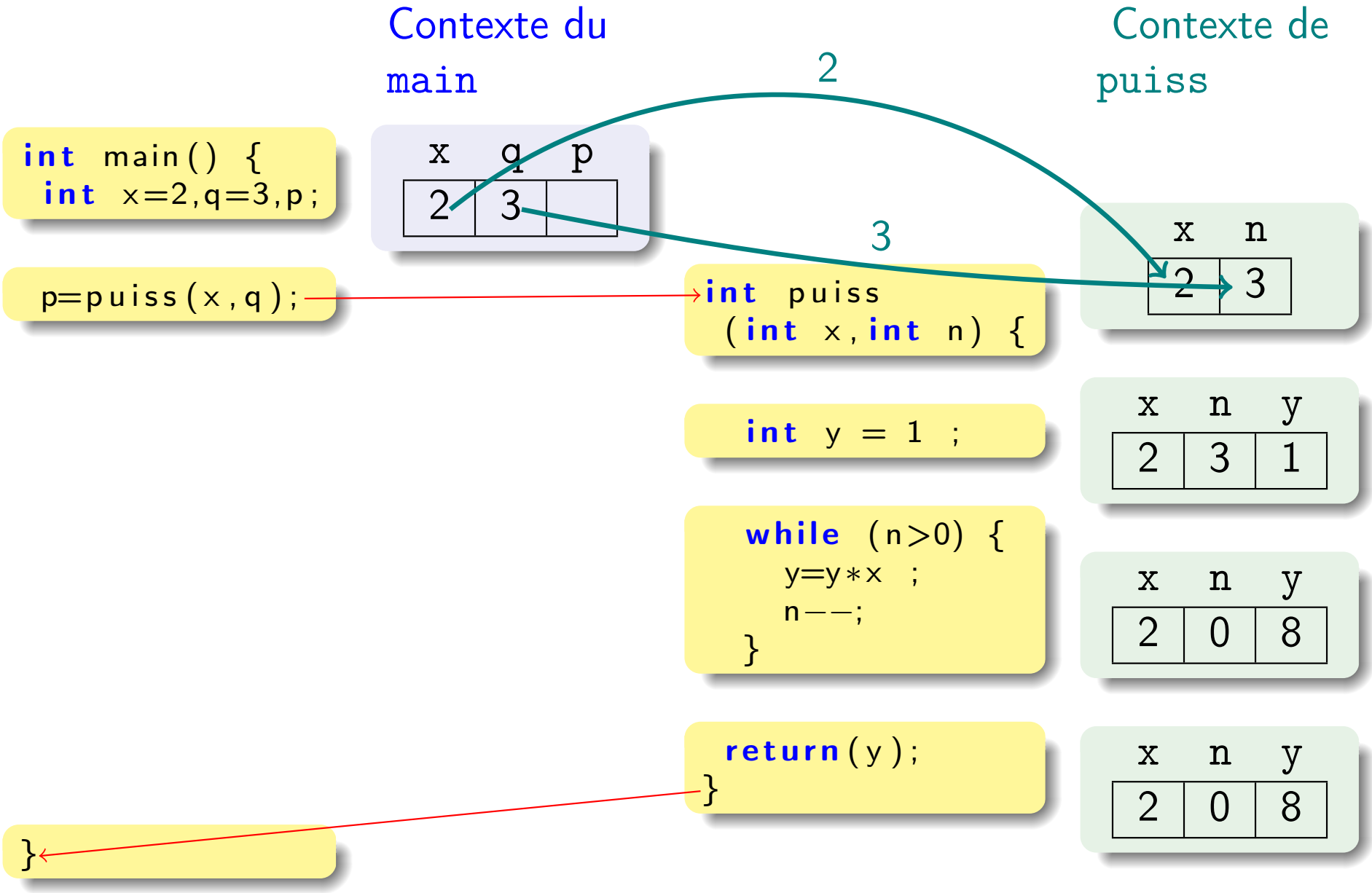
Exemple avec fonction



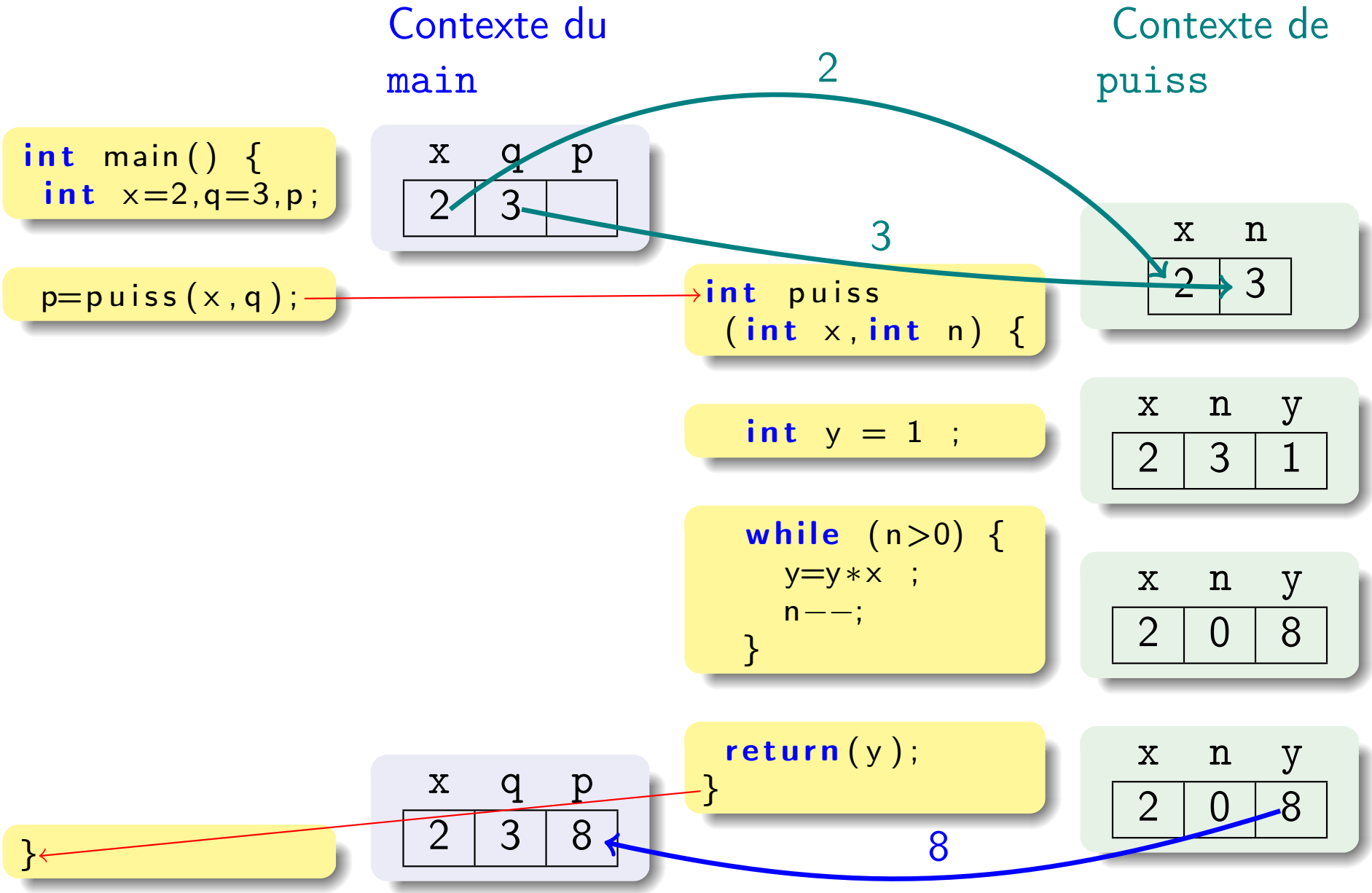
Exemple avec fonction



Exemple avec fonction



Exemple avec fonction



Exemple avec fonction

Contexte du `main`

```
int main() {
 int x=2,q=3,p;
```

x	q	p
2	3	

```
 p=puiss(x,q);
```

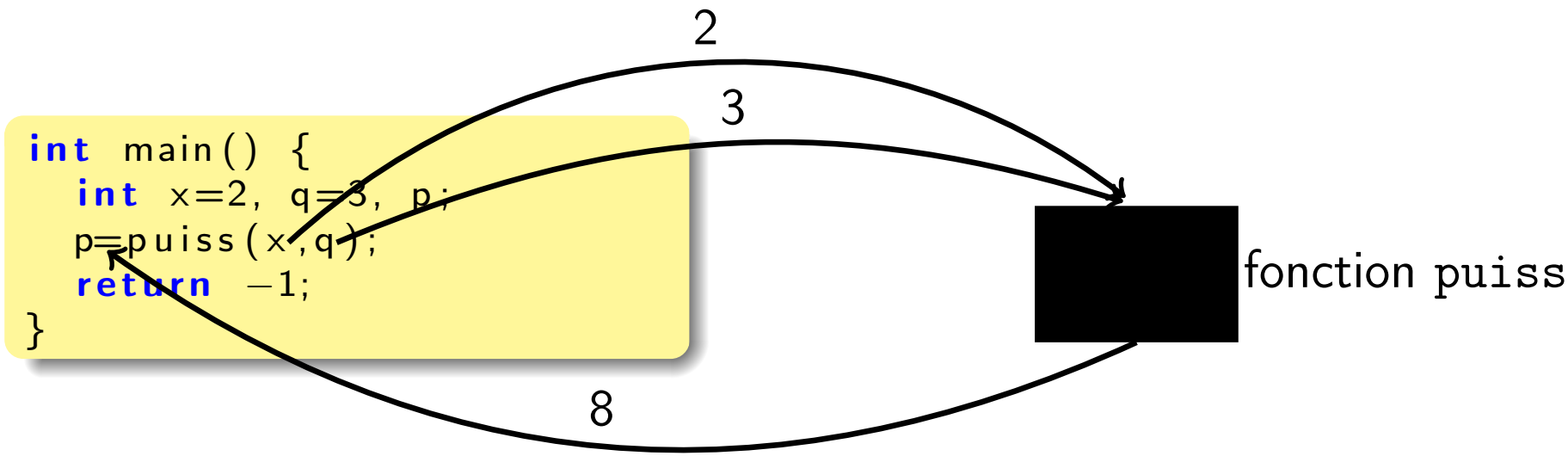
```
int puiss
 (int x,int n) {
  int y = 1 ;
  while (n>0) {
    y=y*x ;
    n--;
  }
 return(y);
}
```

x	q	p
2	3	8

```
}
```

Vision boîte noire

Une fonction peut être vue comme une "boîte noire". On lui donne des entrées, elle renvoie une sortie, et on ne se préoccupe pas de ce qui se passe à l'intérieur.



Passage d'un tableau en argument

- Pour un tableau à une dimension, on passe le tableau et sa taille
  `type_retour nom_fonction(float Tab[N], int n);`
- Pour un tableau à deux dimensions :
  `type_ret nom_fonct(float Tab[N1][N2], int n1, int n2);`

`N`, `N1` et `N2` ne sont pas des variables, ce sont des expressions constantes ou des identificateurs pour le précompileur via un `#define`.

Exemple

```c
#define N1 3
#define N2 2

void affiche(int T[N1][N2], int n1, int n2);

int main() {
  int Tab[N1][N2]={1,2,3,4,5,6};
  affiche(Tab,N1,N2);
  return -1;
}

void affiche(int T[N1][N2], int n1, int n2)
{
  int i,j;
  for (i=0 ; i < n1 ; i++) {
    for (j=0 ; j < n2 ; j++) {
      printf("%d\t",T[i][j]);
    }
    printf("\n");
  }
}
```

Qu'affiche le programme ?

Tolérance

L'indication de la taille de la première dimension est facultatif.

- Tableau à une dimension :
  type_retour nom_fonction(float *Tab*[], int n);

```
void affiche(int T[], int n);
```

- Pour un tableau à deux dimensions :
  type_ret nom_fonct(float *Tab*[][N2], int n1, int n2);

```
void affiche(int T[][2], int n1, int n2);
```

Tableau entrée/sortie

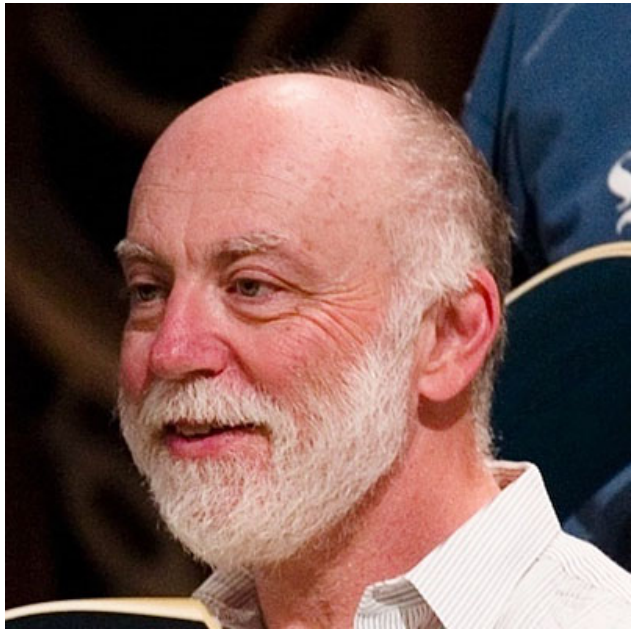
Un tableau est un cas particulier d'argument qui peut être modifié.
C'est un passage par référence (voir cours sur les pointeurs).

Si le tableau est modifié dans la fonction appelée, il sera aussi modifié pour la fonction appelante.

Exemple

```c
#define N 3

void affiche(int Tab[], int n);
void init0 (int Tab[],int n);

int main() {
  int T[N]={1,2,3};
  affiche(T,N);
  init0(T,N);
  affiche(T,N);
  return -1;
}

void init0 (int Tab[],int n)
{
  int i;
  for (i=0 ; i < n ; i++) {
   Tab[i]=0;
  }
}
```

Qu'affiche le programme ?



"To iterate is human, to recurse divine"

L. Peter Deutsch (1946-)

auteur de Ghostscript

Fonction récursive

Un algorithme récursif est un algorithme qui, lors de son exécution, s'appelle lui-même.

Algorithme factorielle

Require: n (entier)
Ensure: nfact (entier)
  if $n = 0$ **then**
    return 1
  else
    return $n \times \text{factorielle}(n-1)$
  end if

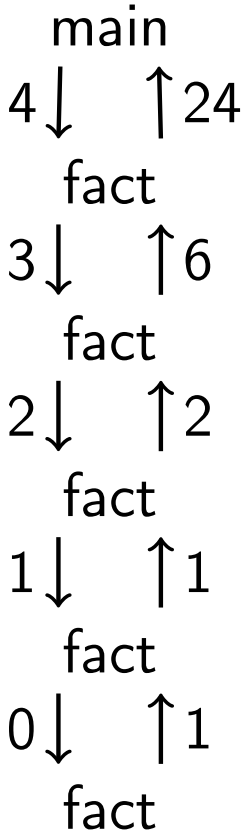
Le langage C permet la récursivité avec les fonctions :

```
int fact (int n) {
  if (n==0) return (1) ;
  else return(n * fact (n-1));
}
```

Contexte d'exécution

Chaque appel à la fonction crée son propre contexte d'exécution (peut être gourmand en mémoire).

```
int fact (int n) {
  if (n==0) return (1) ;
  else return(n * fact (n-1));
}

int main() {
  int z ;
  z = fact(4);
  return -1;
}
```



Fonctions récursives

Selon les problème traités, la récursivité permet d'obtenir un code source succinct.

Par contre, quand elle est mal réalisée :

- Le code peut tourner sans fin ;
- La mémoire utilisée peut être énorme (à cause de la multiplication des contextes d'exécution) ;
- Cela peut ralentir l'exécution du programme.

A utiliser avec précaution.

La fonction rand

Prototype

```
int rand();
```

- Renvoie un nombre aléatoire entre 0 et RAND_MAX.
- Utilisation :

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

int main() {
  int alea ;
  alea = rand() ;
  printf("%d",alea);
  return -1;
}
```

Valeur type de RAND_MAX :
2 147 483 647

Dans cet exemple, la valeur renvoyée par rand sera la même à chaque exécution du programme

Comment changer de nombre aléatoire à chaque exécution ?

La solution

Il faut initialiser de manière différente le générateur aléatoire à chaque exécution.
Astuce : Utiliser l'heure d'exécution.

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

int main() {
  int alea ;
  srand(time(NULL));
  alea = rand() ;
  printf("%d",alea);
  return -1;
}
```

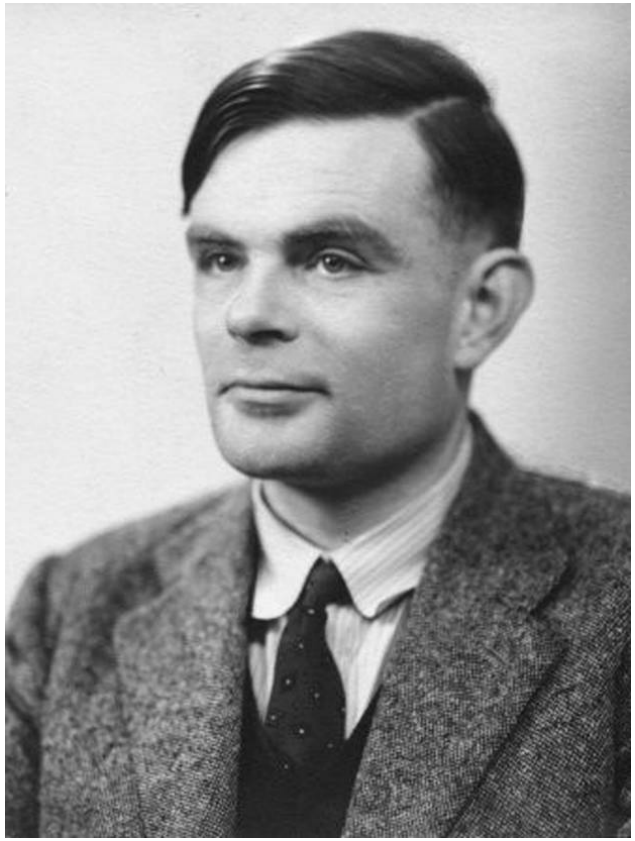
Les fonctions mathématiques `math.h`

```
#include <math.h>
```

Fonctions trigonométriques			Fonctions élémentaires		
`sin(x)`	`cos(x)`	`tan(x)`	`exp(x)`	`pow(x,y)`	`sqrt(x)`
`asin(x)`	`acos(x)`	`atan(x)`	`log(x)`	`ceil(x)`	`floor(x)`
`sinh(x)`	`cosh(x)`	`tanh(x)`	`log10(x)`	`fabs(x)`	`fmod(x,y)`

Nécessite l'option `-lm` à la compilation

```
gcc -o prg prg.c -lm
```



"A computer would deserve to be called intelligent if it could deceive a human into believing that it was human."

Alan Turing (1912-1954)

Un peu d'histoire...

1900 : Problèmes d'Hilbert (2ème problème : Est-ce que l'arithmétique est cohérente ?)

1931 : Gödel démontre que l'arithmétique est incomplète. Cela est lié au problème de la décision.

1936 : Introduction de la machine de Turing par Alan Turing à l'âge de 24 ans pour montrer que certains problèmes ne peuvent pas être résolus par un algorithme (c'est à dire par la machine de Turing).

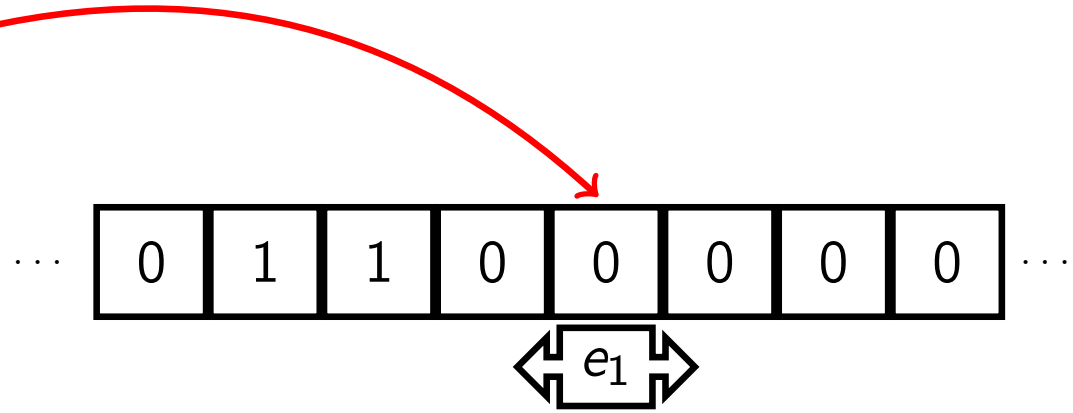
1942 : Participation au décodage d'Enigma.

1946 : Création de l'ENIAC (Electronic Numerical Integrator and Computer).

1954 : Mort de Turing. 

Définition de la machine de Turing

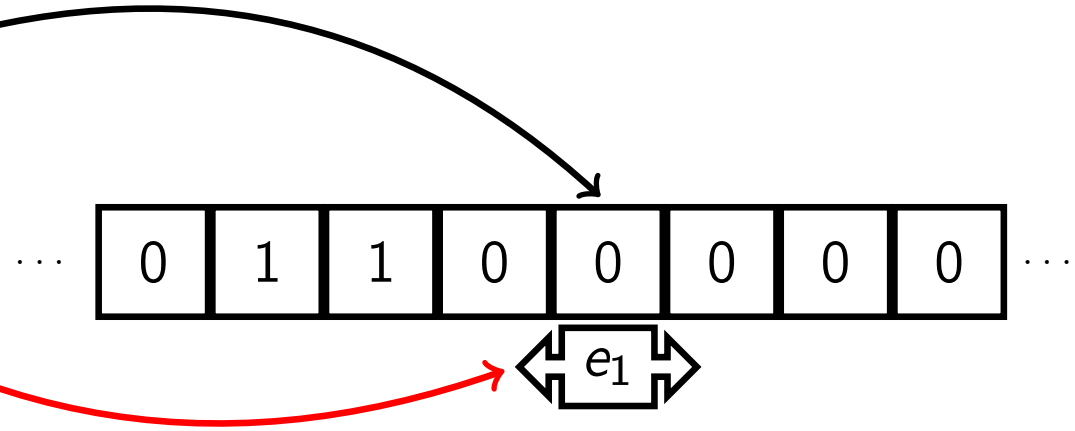
1. Un ruban infini
   contenant des symboles.



Ancien état	Symbole lu	Symbole écrit	Mouvement	Nouvel état

Définition de la machine de Turing

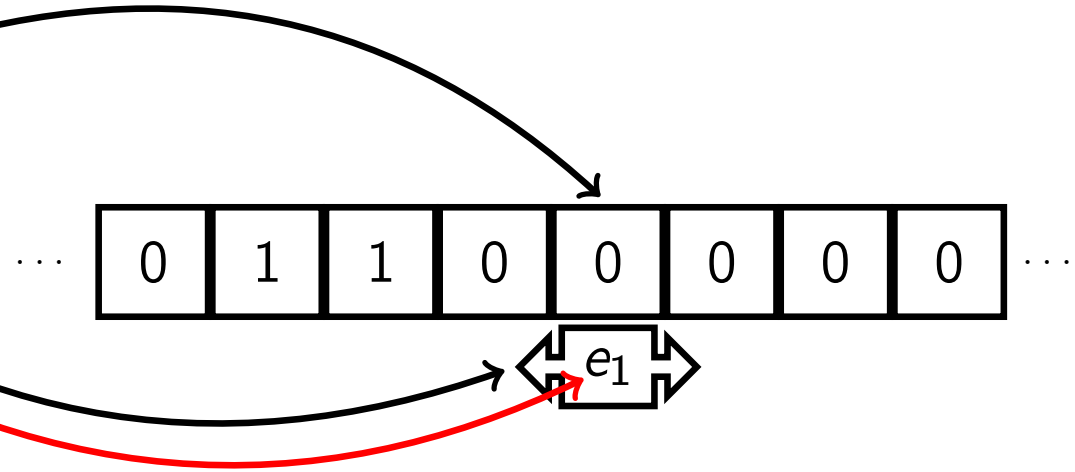
1. Un ruban infini contenant des symboles.
2. Un tête de lecture/écriture.



Ancien état	Symbole lu	Symbole écrit	Mouvement	Nouvel état

Définition de la machine de Turing

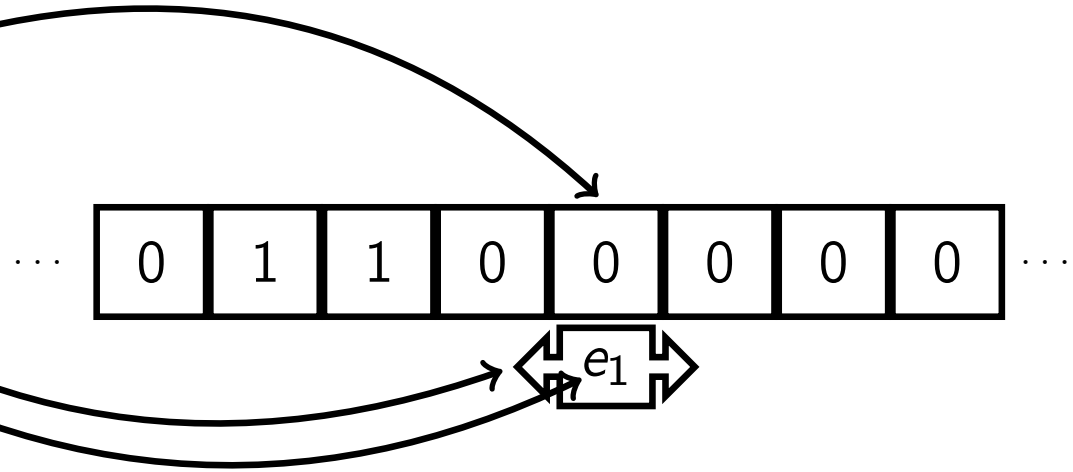
1. Un ruban infini contenant des symboles.
2. Un tête de lecture/écriture.
3. Un registre d'état.



Ancien état	Symbole lu	Symbole écrit	Mouvement	Nouvel état

Définition de la machine de Turing

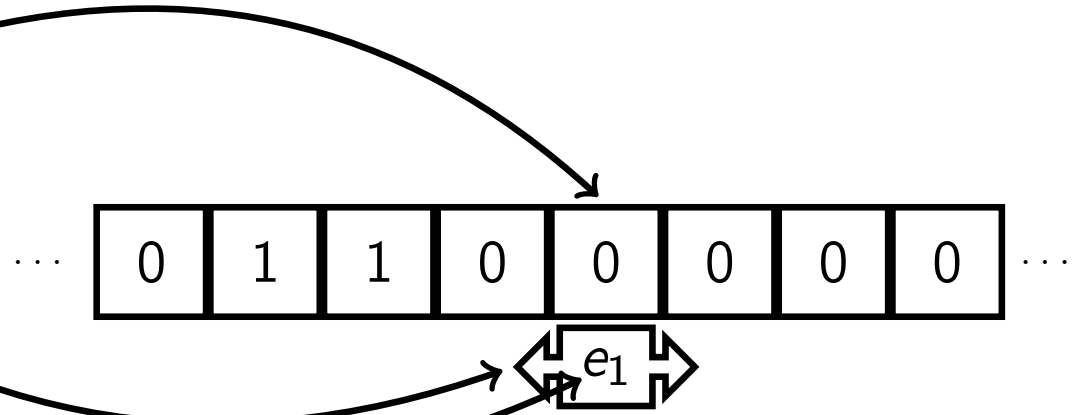
1. Un ruban infini contenant des symboles.
2. Un tête de lecture/écriture.
3. Un registre d'état.
4. Une table d'actions.



Ancien état	Symbole lu	Symbole écrit	Mouvement	Nouvel état

Définition de la machine de Turing

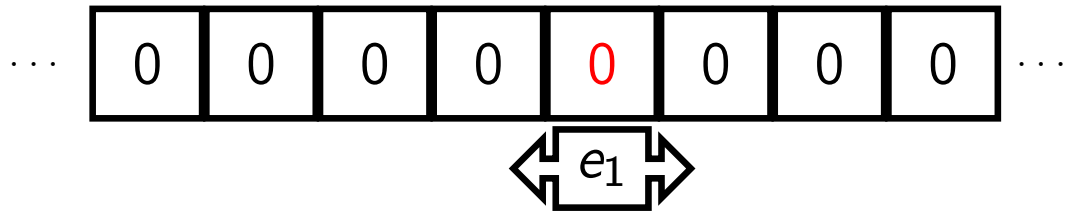
1. Un ruban infini contenant des symboles.
2. Un tête de lecture/écriture.
3. Un registre d'état.
4. Une table d'actions.
5. Un état et une position de départ sur le ruban.



Ancien état	Symbole lu	Symbole écrit	Mouvement	Nouvel état

Fonctionnement d'une machine de Turing

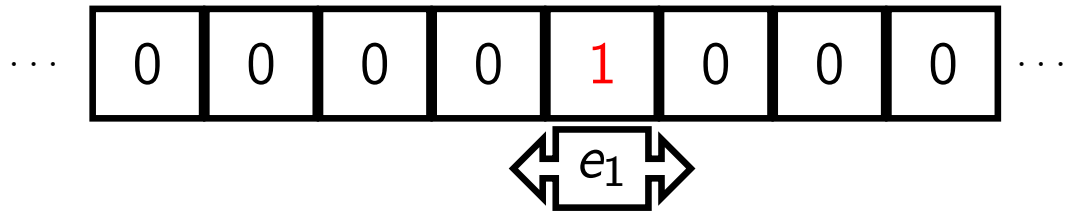
1. La tête de lecture lit un symbole s.



Ancien état	Symbole lu	Symbole écrit	Mouvement	Nouvel état
e_1	0	1	D	e_2
e_2	0	1	D	e_3
e_3	0	1	D	e_4
e_4	0	1	D	e_5
e_5	0	1	G	e_4
e_1	1	0	G	ARRET
e_2	1	1	G	e_1
e_3	1	0	G	e_2
e_4	1	0	G	e_3
e_5	1	0	D	e_4

Fonctionnement d'une machine de Turing

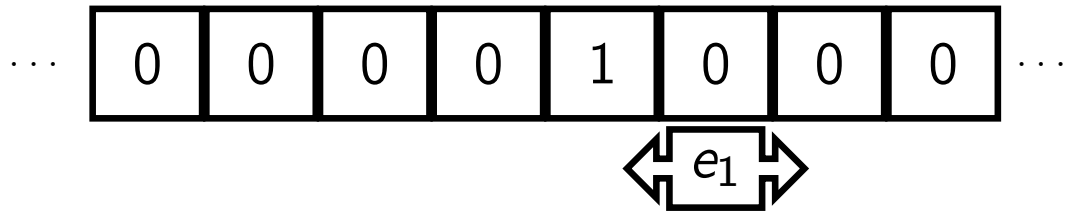
1. La tête de lecture lit un symbole s.
2. En fonction de s et de son état, la tête écrit un symbole s'



Ancien état	Symbole lu	Symbole écrit	Mouvement	Nouvel état
e_1	0	1	D	e_2
e_2	0	1	D	e_3
e_3	0	1	D	e_4
e_4	0	1	D	e_5
e_5	0	1	G	e_4
e_1	1	0	G	ARRET
e_2	1	1	G	e_1
e_3	1	0	G	e_2
e_4	1	0	G	e_3
e_5	1	0	D	e_4

Fonctionnement d'une machine de Turing

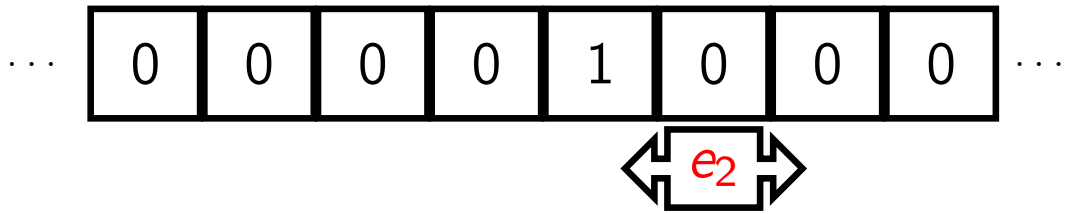
1. La tête de lecture lit un symbole s.
2. En fonction de s et de son état, la tête écrit un symbole s'
3. En fonction de son état la tête se déplace d'un cran.



Ancien état	Symbole lu	Symbole écrit	Mouvement	Nouvel état
e_1	0	1	D	e_2
e_2	0	1	D	e_3
e_3	0	1	D	e_4
e_4	0	1	D	e_5
e_5	0	1	G	e_4
e_1	1	0	G	ARRET
e_2	1	1	G	e_1
e_3	1	0	G	e_2
e_4	1	0	G	e_3
e_5	1	0	D	e_4

Fonctionnement d'une machine de Turing

1. La tête de lecture lit un symbole s.
2. En fonction de s et de son état, la tête écrit un symbole s'
3. En fonction de son état la tête se déplace d'un cran.
4. La machine change d'état.



Ancien état	Symbole lu	Symbole écrit	Mouvement	Nouvel état
e_1	0	1	D	e_2
e_2	0	1	D	e_3
e_3	0	1	D	e_4
e_4	0	1	D	e_5
e_5	0	1	G	e_4
e_1	1	0	G	ARRET
e_2	1	1	G	e_1
e_3	1	0	G	e_2
e_4	1	0	G	e_3
e_5	1	0	D	e_4