Les boucles et les tableaux

A. Lambert/P. Courtieu

DSP - USAL 34

Les instructions complexes (boucles)

La boucle `while`

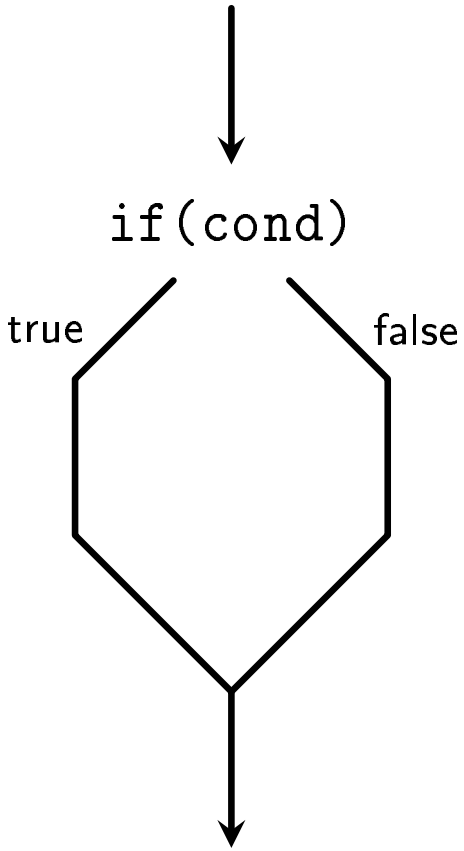
Nous avons vu qu'un programme est une séquence d'instruction, avec la possibilité de tester des conditions (`if`).

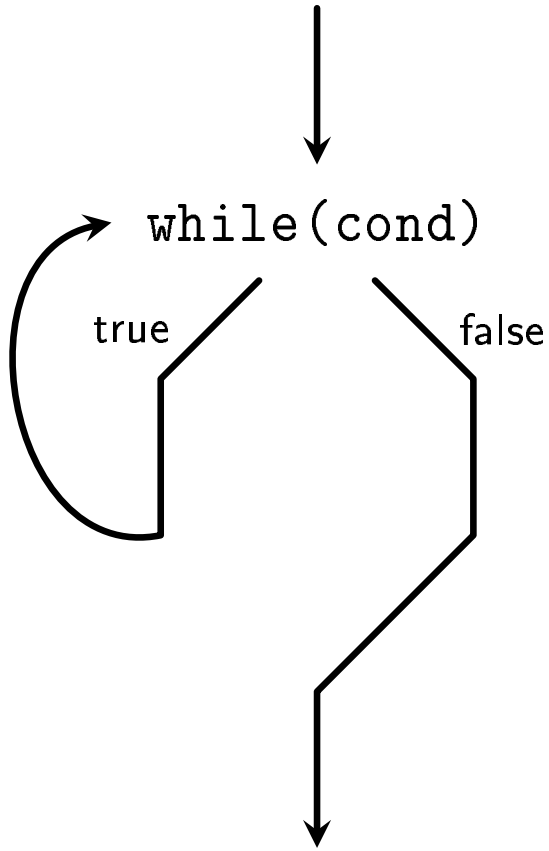
Comment répéter (*itérer*) un traitement plusieurs fois, sachant que ?

- le nombre d'itérations varie d'une exécution à l'autre ?
- quand arrêter les itérations ?

  Quand le calcul répétitif est fini = lorsqu'une condition devient fausse.

Flot des instructions

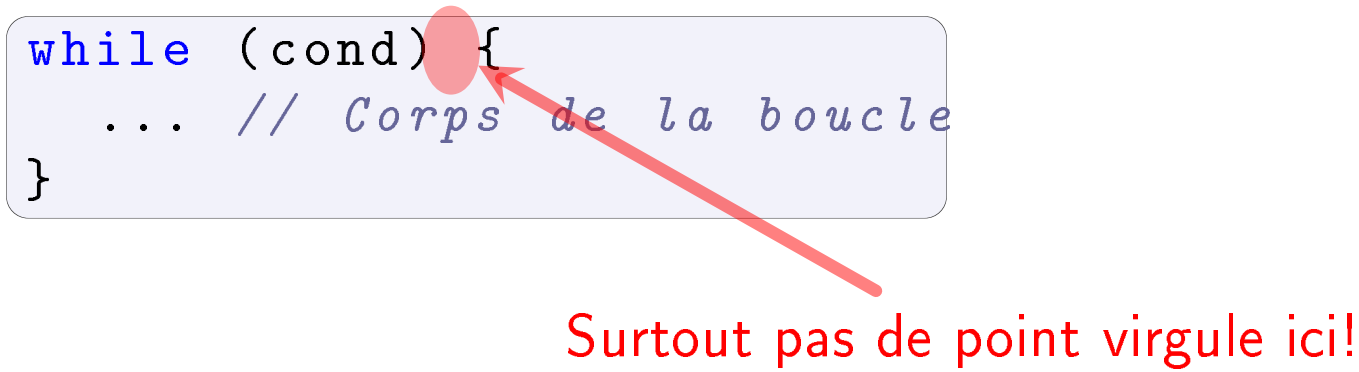




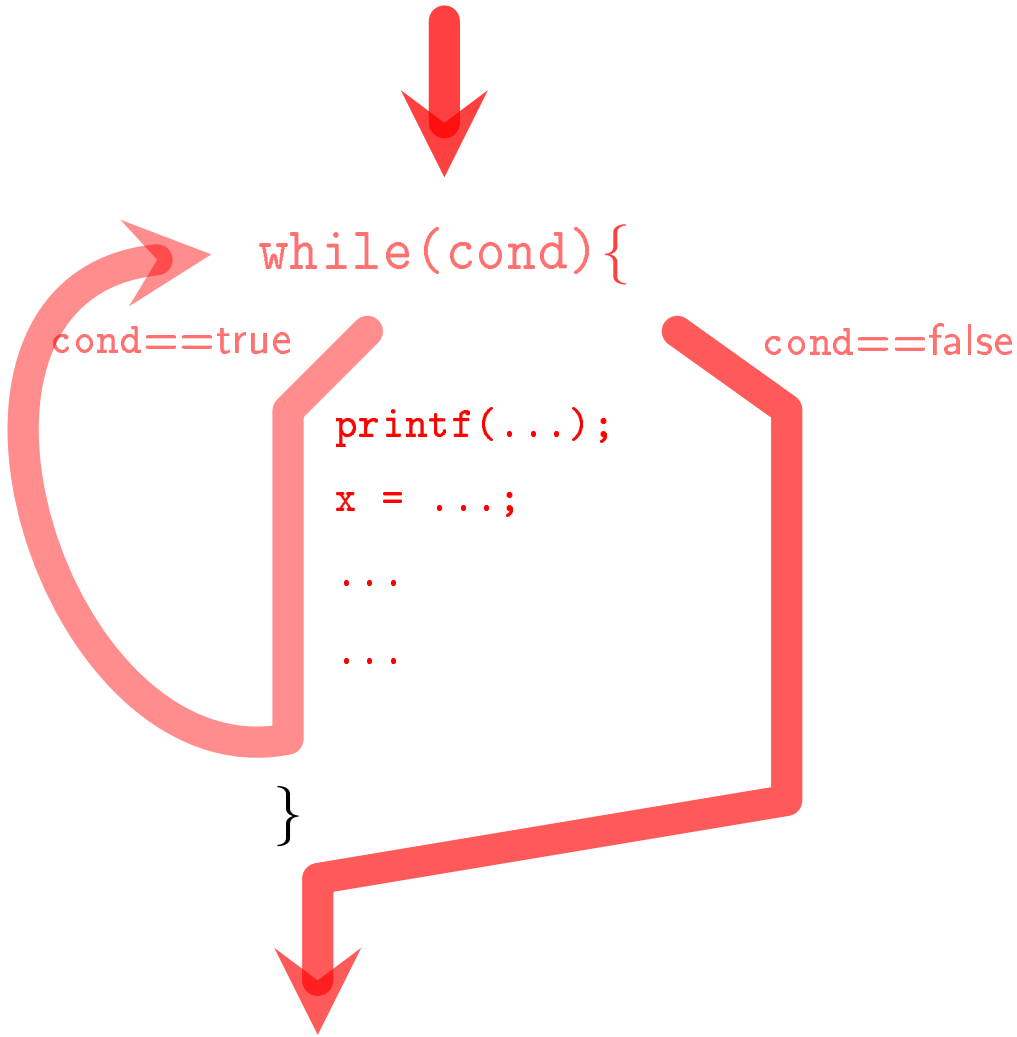
La boucle `while`

Syntaxe comparable à un `if` sans `else` :

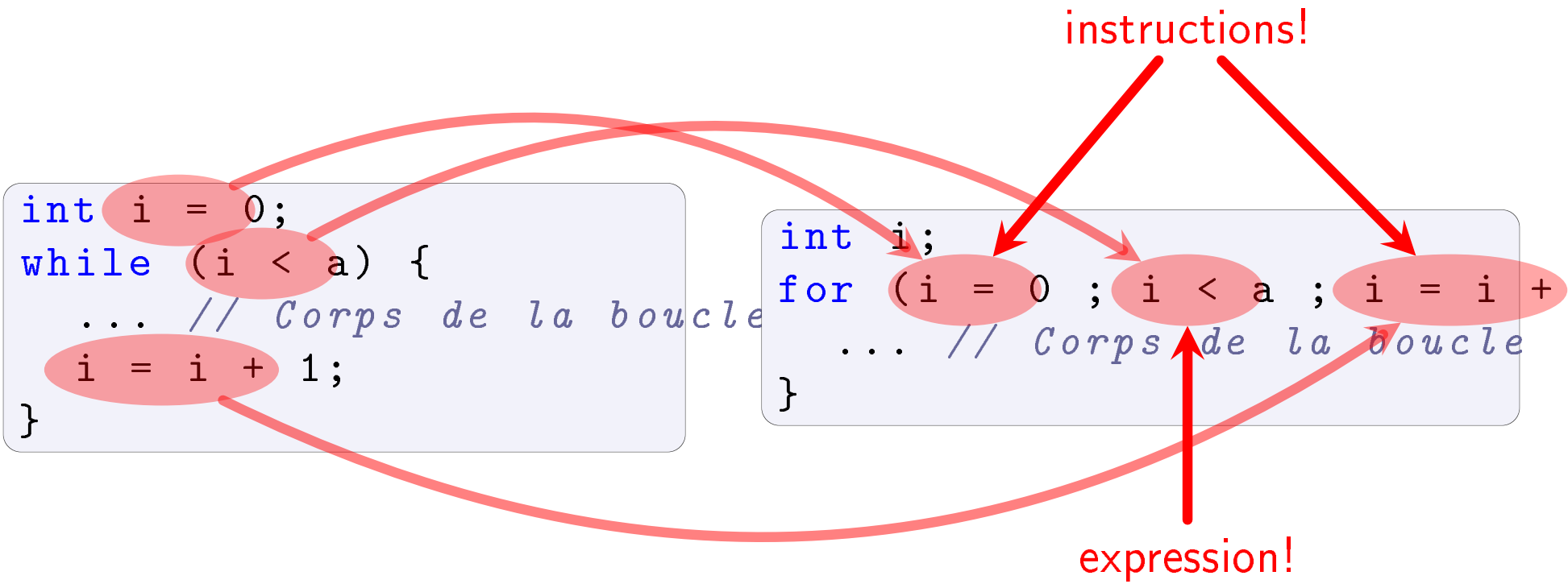
```
while (cond) {
   ... // Corps de la boucle
}
```



Flot des instructions



La boucle `for` – un cas particulier du `while`



`for` Réservée aux boucles avec manipulations *simples* d'indices

La condition

`cond` condition booléenne quelconque (comme pour un `if`)

- `< > <= ==`... opérateurs de comparaison numérique. ex : `x < 0`
- `&& || !`... opérateurs booléens ex : `x < 0 || x > 9`

```
while (x < 8 && !(y == f(z)) || u) {
   ... // Corps de la boucle
}
```

Les tableaux

Notion de tableau

- Pour pouvoir écrire des programmes un peu plus intéressants, il est nécessaire d'appliquer des traitements à des séquences de données. Une manière classique est de les rassembler dans un **tableau**.

- Dans l'exemple qui suit, on réunit dans un tableau le nombre d'heures d'ensoleillement de chaque jour de la semaine.

4	8	2	6	9	3	1
0	1	2	3	4	5	6

Les **indices du tableau** représentent les jours de la semaine.

Les tableaux

- Un **tableau** est une **structure de données** qui réunit des valeurs (données) d'un **même type** (le type `int` dans l'exemple).
- C'est une suite de cases contiguës repérées (indicées) par des entiers (`int`), en partant `0`.
- Un tableau constitue une nouvelle valeur qui doit être typée.
- Dans l'exemple, le **type** est `int[]`, les indices de `0` à `6` correspondent aux jours de la semaine.
- On peut alors déclarer une variable (`uneSemaine`) de ce nouveau type, enregistrer des valeurs dans les cases de ce tableau, et sélectionner une case connaissant son indice (`uneSemaine[i]`).

Les tableaux en C

- C'est une variable qui contient *plusieurs valeurs* d'un même type.
- Si il a `n` valeurs, elles ont *numérotées de* 0 *à* $n-1$ 
- la i^e case d'un tableau `t`, (notée `t[i]`) = équivalente à une variable.
  - Affectation : `t[i] = 12 + f(x);`
  - Utilisation : `x = (t[i] - 1) * u[j];`
- Déclaration :

```
int t [9]; // 9 cases
int t = {9,8,7,6,5,4,3,2,1}; //9 cases
```

Parcours d'un tableau

```
int t [9]; // 9 cases
int t = {9,8,7,6,5,4,3,2,1}; //9 cases
int idx = 0;
while(idx<9) {
  t[idx] = 0;
  idx = idx + 1;
}
```

longueur-1 est l'indice de la dernière case

0	0	0	0	0	0	0	0	0

```
for(idx=0; idx<9 ; idx = idx + 1){
  t[idx] = idx;
}
```

0	1	2	3	4	5	6	7	8

Passer un tableau en paramètre d'une fonction

Spécificité du C :

- En C : un tableau ne « connaît »pas sa taille.
- solution : « transporter »la taille en même temps que le tableau.
- argument supplémentaire dans les fonctions :

```
void f (int t[] , int length_t){
   ...
}

int main() {
   int t[7];
   f(t,7);
}
```

Pattern classiques : « il existe »

Déterminer si *il existe* une case `i` telle que `test(t[i])` vrai.

la réponse « pour l'instant »

```
int i = 0;
int candidatReponse = FALSE;
while(i<9) {
  if(test(t[i])) {
    candidatReponse = TRUE;
  }
  i = i + 1;
}
```

pour l'instant pas de cases OK

basculement (définitif) du booléen

ne pas remettre à false

candidatReponse = réponse à la question

- Remarque : si 0 élément réponse `FALSE`

Pattern classiques : « pour tout »

déterminer si *toutes* les cases `i` telles que un `test(t[i])` est vrai.

```
int i = 0;
int candidatReponse = TRUE;
while(i<9) {
  if(! test(t[i])) {
    candidatReponse=FALSE;
  }
  i = i + 1;
}
```

pour l'instant toutes les cases OK

basculement (définitif) du booléen

ne pas remettre à true

candidatReponse = réponse à la question

- Remarque : si 0 élément réponse `TRUE`

Pattern classiques : « meilleure case »

déterminer *la meilleure* case `i` pour `compare(t[i],t[j])`

```
int i = 0;
int candidatReponse = t[0];
while(i<9) {
  if(compare(t[i],candidatReponse)) { // t[i] meilleur
    candidatReponse = t[i];
  }
  i = i + 1;
}
```

Pour l'instant meilleur candidat (0,

remplacement du candidat

- Remarque : *pas de sens* si 0 élément
- Remarque : `i` peut démarrer à 1

Retour sur les types

Rappel : le typage

```
int x;
x = 8;
... (x + 9) ...;
... f(x) ...;        /* int f (int a) */
```

- Chaque variable est déclarée avec son type `T` (`int`, `char`, ...)
- Affectation : seulement par une valeur du type `T`
- Opérateur : seulement si `T` est le bon type
- Fonction : seulement si `T` est le bon type

$\Rightarrow$ Sécurité, pas d'erreur de type à l'exécution

MAIS : conversion entre type (pratique mais source d'erreurs)

Coercions implicites (1/4)

```
int x;
x = 8.3;
... (x + 9.2) ...;
... f(12.5) ...;  /* int f (int a) */
```

- Entorse aux règles de typage
- Certains types sont convertibles entre eux (*coercion*)
-  conversion *implicite* malgré perte de précision éventuelle !

Coercions implicites (2/4)

```
int x;
x = 8.3;
... (x + 9.2) ...;
... f(12.5) ...;  /* int f (int a) */
```

affectation : conversion implicite vers le type de la variable

`x` $\leftarrow$ 8 $\leftarrow$ 8.3

opérateur : conversion des deux arguments vers un même type, plus grand ou égal aux deux types de départ

`x:int` mais `9.2:float` donc on traduit tout vers `float`

`x=8` $\leftarrow$ 8.0 $+$ 9.2 $\rightsquigarrow$ 17.5

fonction : arguments convertis vers le type attendu par la fonction

Coercions implicites (3/4)

```
int x;
x = 8.9 + 9.2;
```

- ?
- x ←18

Coercions implicites (4/4)

```
int x=8;
double y = 2.3;
x = x/y;
```

- ?
- x ←3

Coercions implicites (4/4)

```
int x=8.3;
double y = 2.3;
x = x/y;
```

- ?
- x $\leftarrow$3

Coercion : priorités

```
int n = 8;
double x = 2.6;
y = n / 3 + x;
printf ("n = %d , x = %fd, y = %fd\n",n,x,y);
```

- `n = 8 , x = 2.600000d, y = 4.600000d`
- `n / 3` $\rightsquigarrow$ 2 `int`
- $2 + 2.6 \rightsquigarrow 4.6$

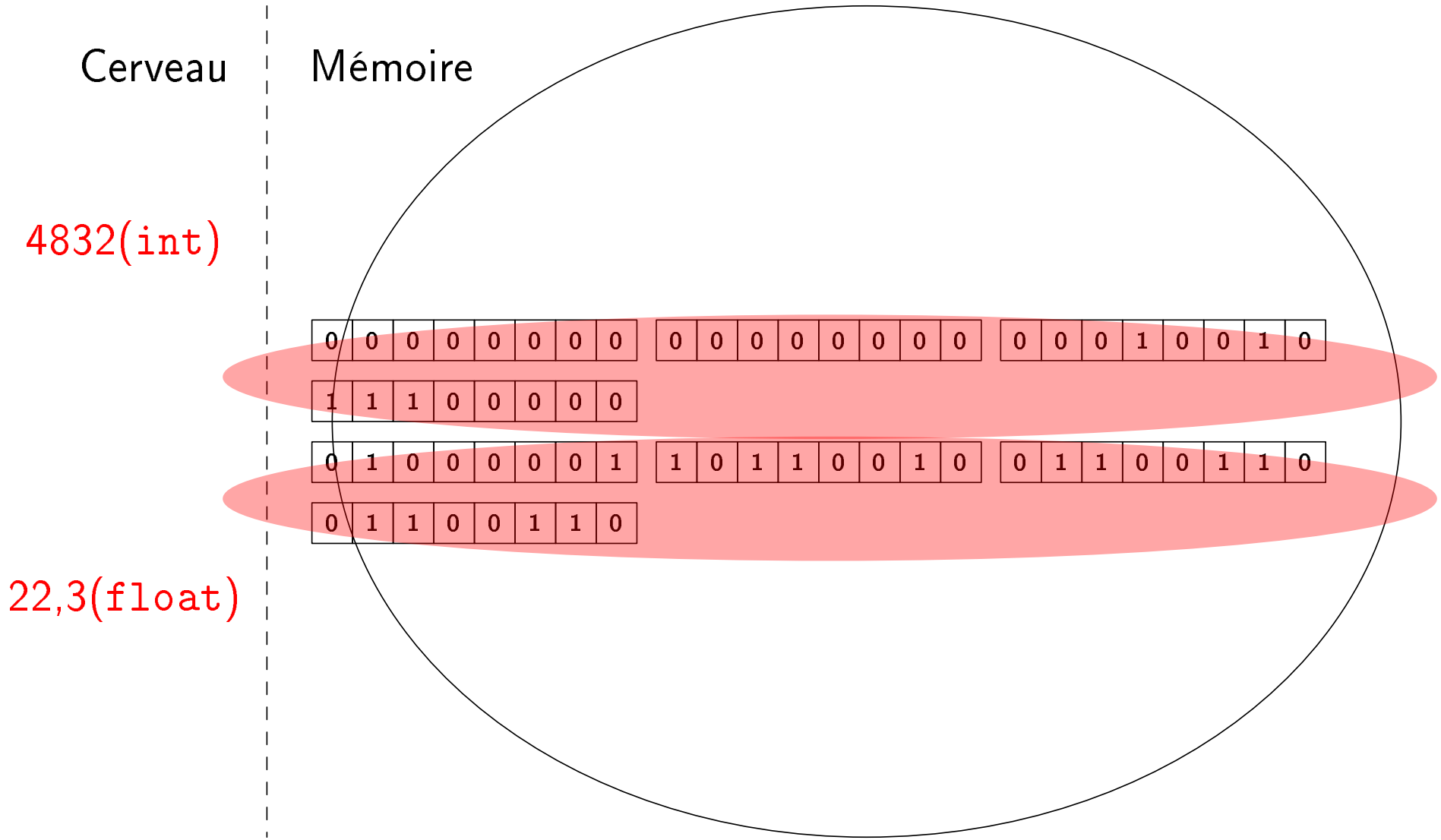
Représentation des données en mémoire

Donnée vues $\neq$ Données mémoires

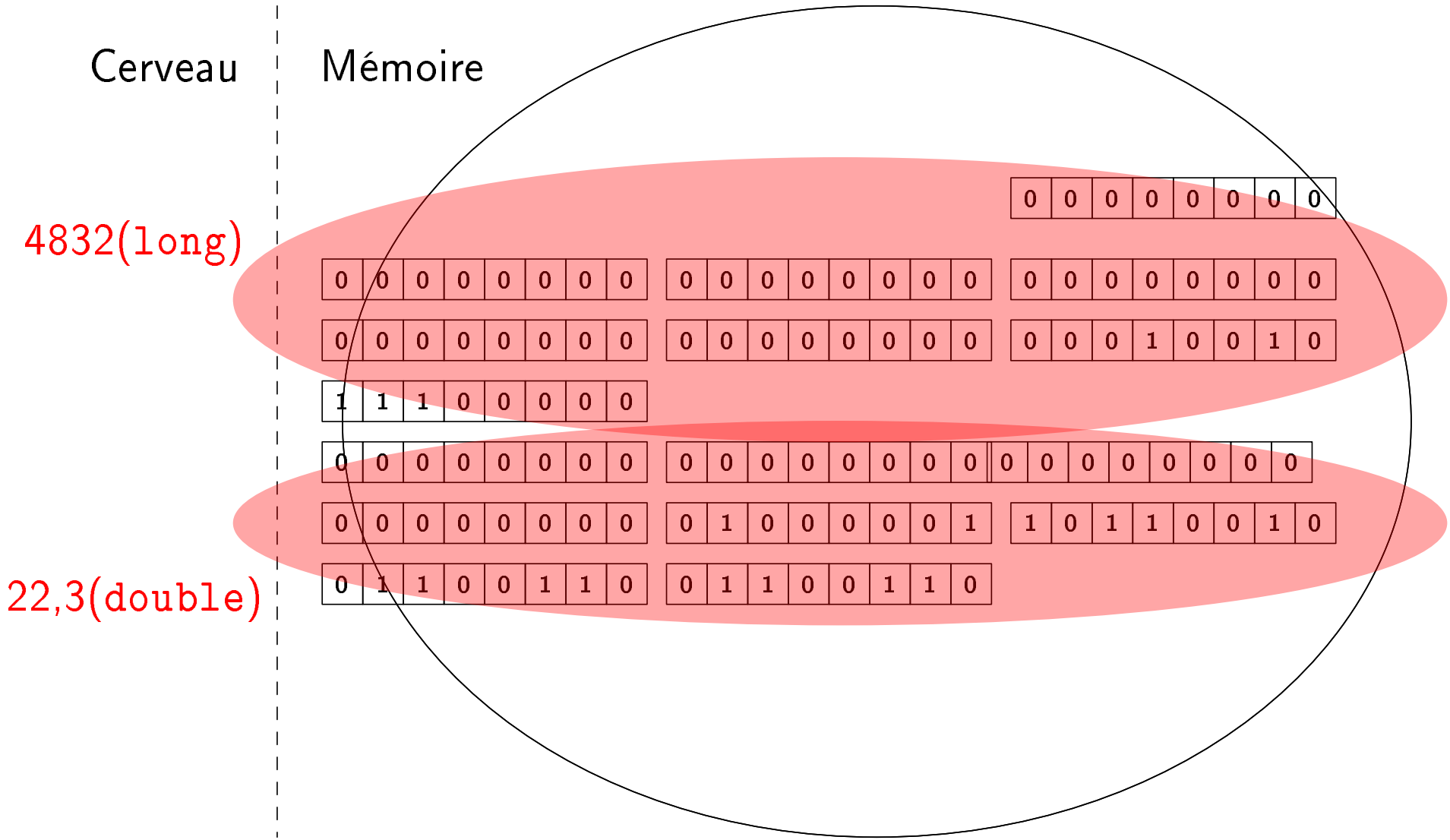
- Données pensées par l'utilisateur :
  - entiers, réels
  - textes
  - « grille » du morpion

- Données dans la mémoire de l'ordinateur :
  - des octets
  - des octets
  - des octets

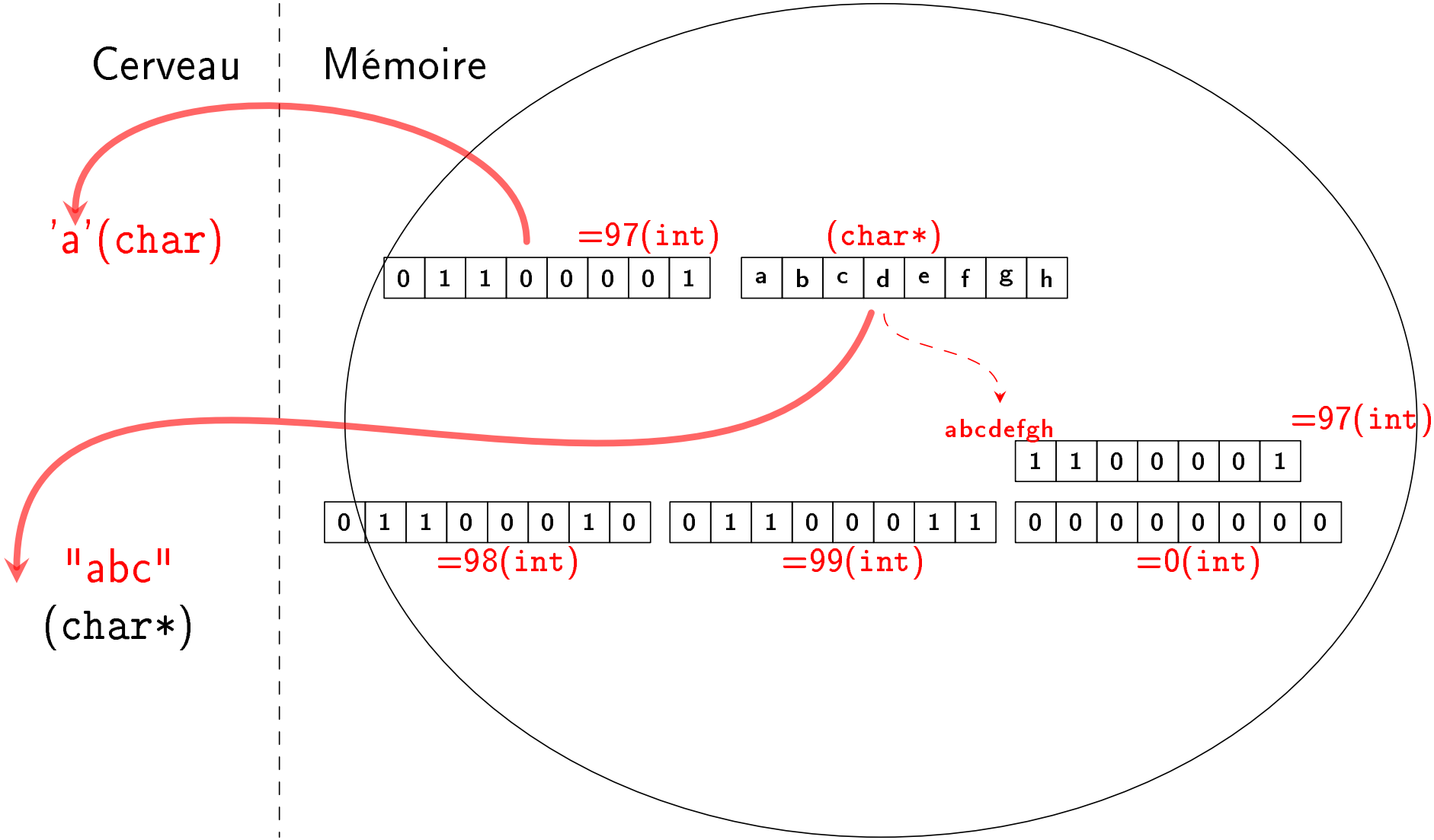
Représentation Mémoire – Les nombres



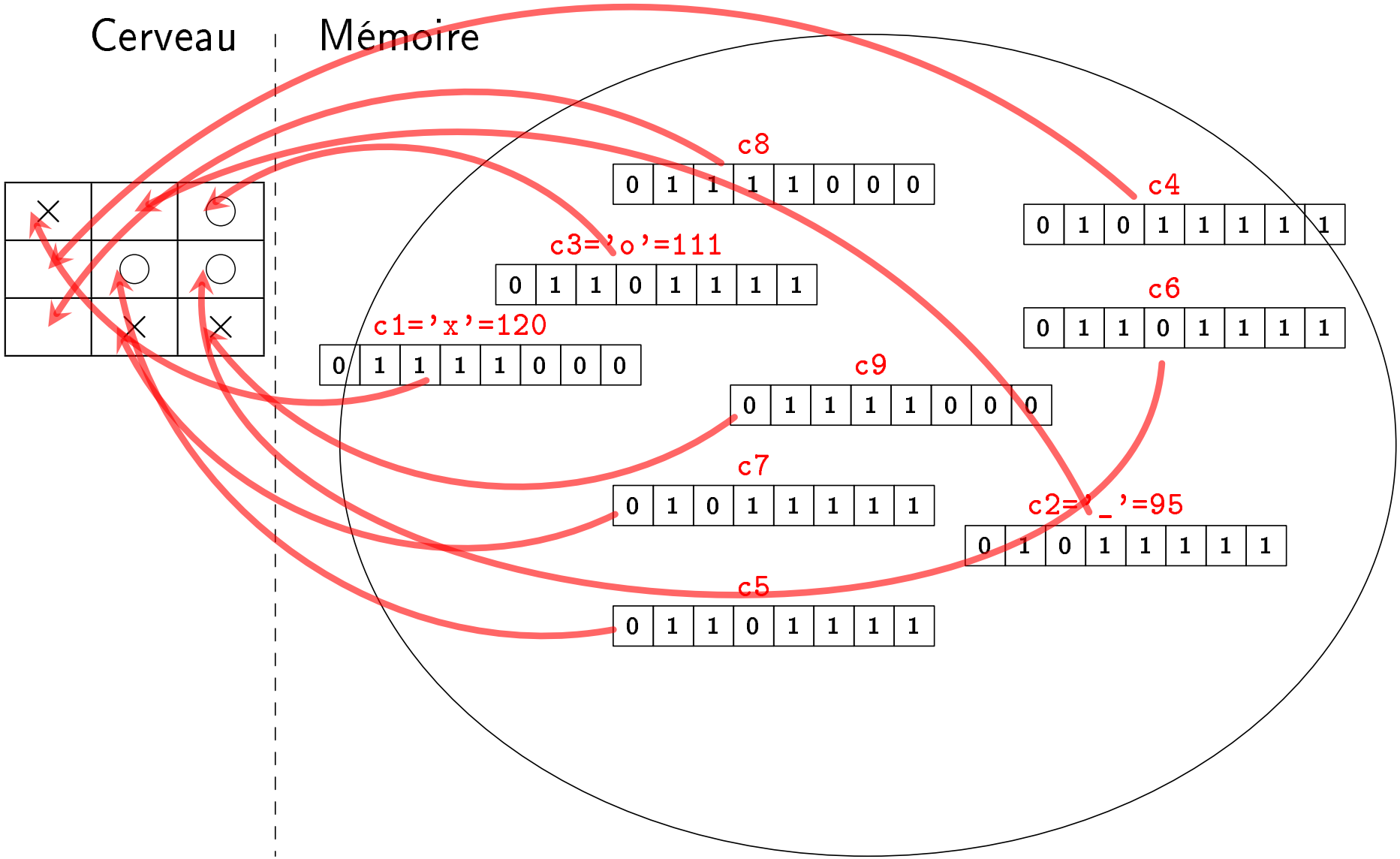
Représentation Mémoire – Les nombres



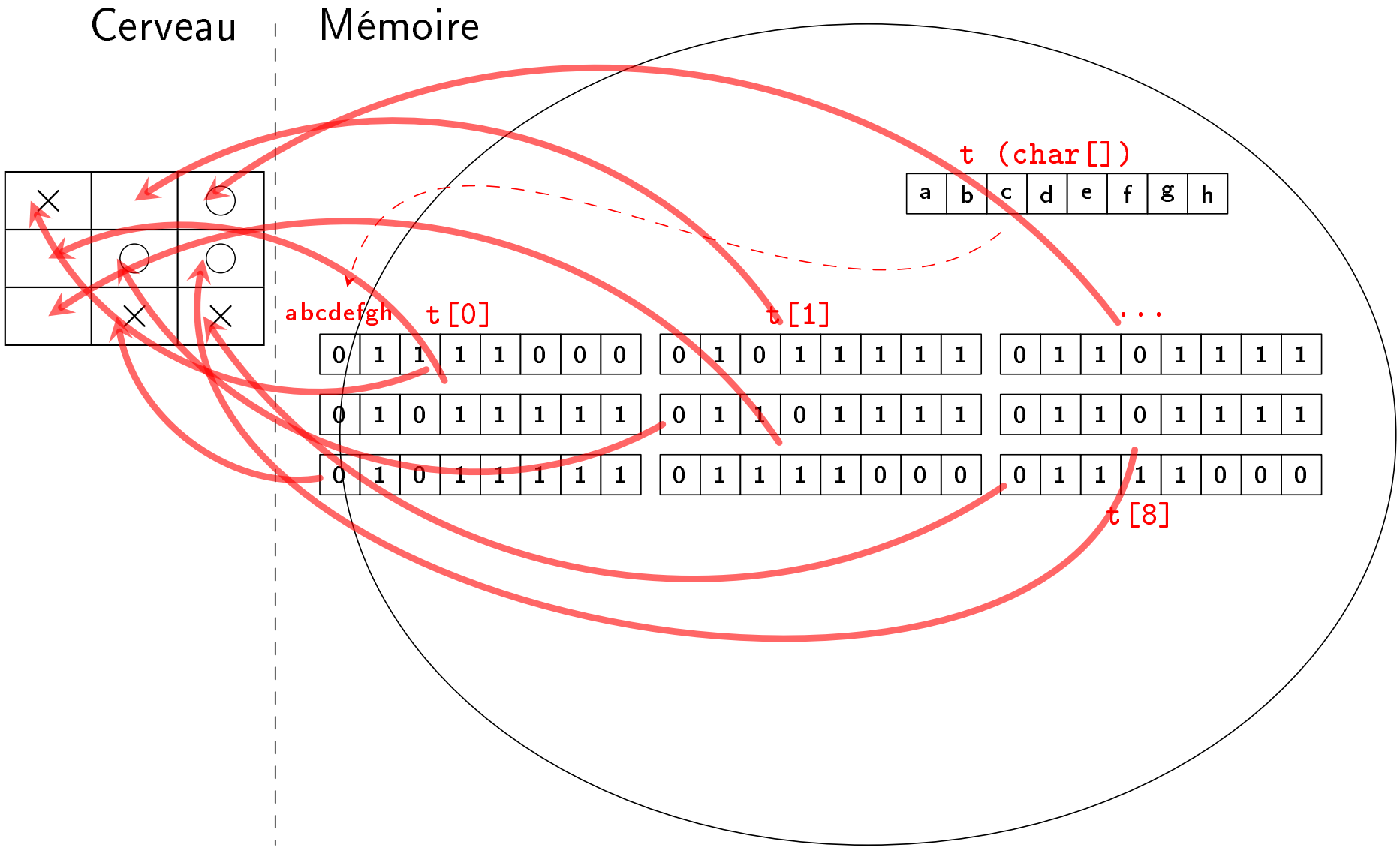
Représentation Mémoire – Les caractères



Représentation Mémoire – la grille du morpion (1/2)



Représentation Mémoire – La grille du morpion (2/2)



Représentation Mémoire

- Ce qui est important :
  la cohérence entre représentation mémoire et algorithmes

- programmer = choisir une représentation mémoire pertinente
  + concevoir les algorithmes pour la manipuler

Représentation Mémoire

Critère de pertinence	`c1...c9`	`char t[9]`
Correcte (fidèle)	OK	OK
Complète (toutes les informations)	OK	OK
Pratique (facile à manipuler)	Bof	OK
Efficace (en taille et en calcul)	OK	OK

Une seule variable veut dire :

- manipulation du numéro de case via l'indice
- meilleur passage à l'échelle (30×30...)
- futures évolutions (puissance 4, bataille navale etc)
- fonctions utiles pour d'autre jeux (`voisinDroite`, `voisinHaut`, etc)