Information, Calcul et Communication

Composante Pratique: Programmation C++

MOOC semaine1: bases de programmation

Les types des données et des constantes littérales

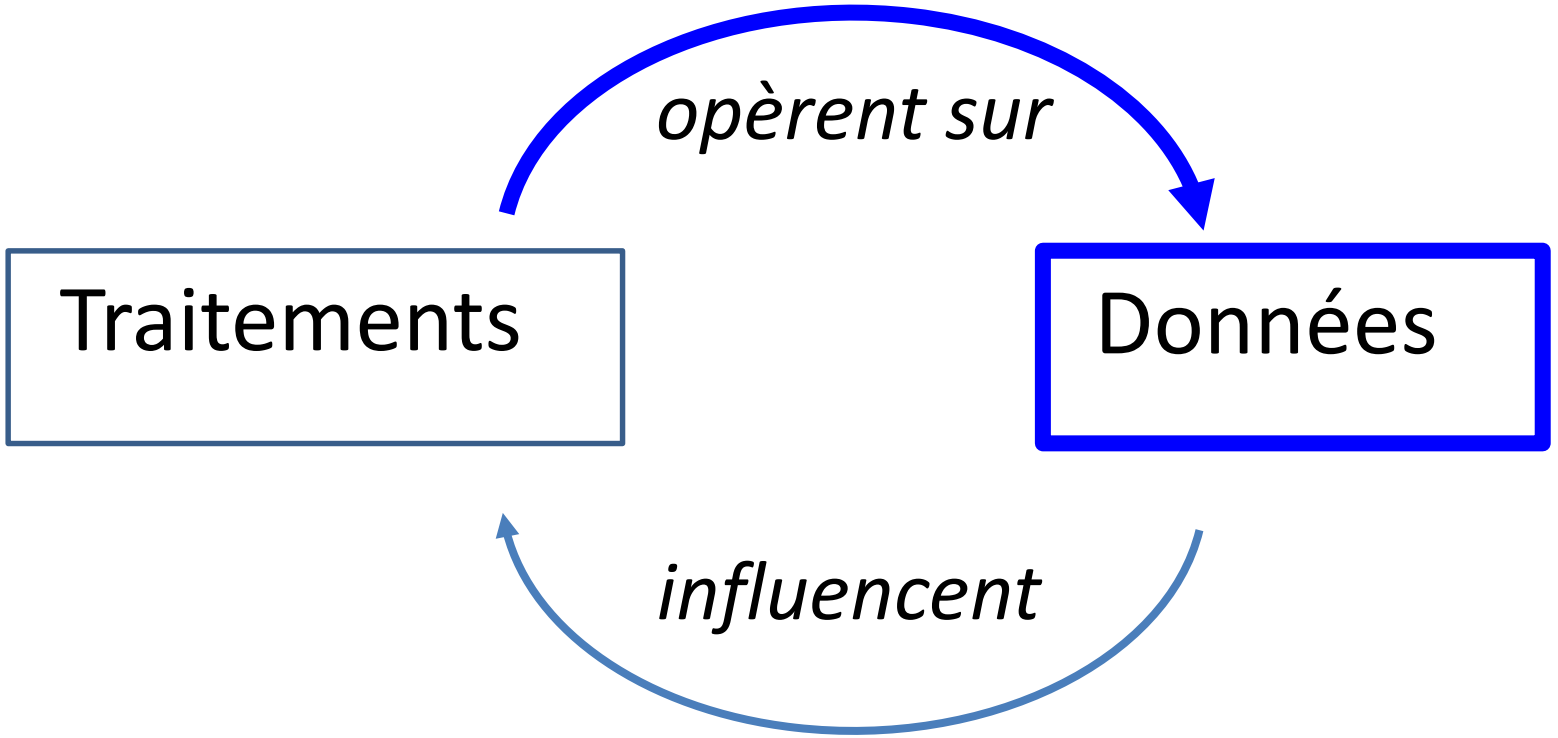
Organisation de la mémoire centrale

Variables, opérateurs et expressions

Constantes littérales et magic numbers

Cycle de développement

Aujourd'hui l'accent est mis sur les **données**,
leur représentation à l'aide de **type** de base,
et leur **mémorisation** dans des **variables**



Nos premiers pas en matière de **traitement** vont porter sur les **opérateurs** manipulant la valeur des **données** au sein **d'expressions du langage C++**

Représentation des données en C++

Nous avons vu en ICC-théorie des méthodes de représentations différentes pour les **nombres entiers** et pour les **nombres à virgule**.

Un processeur va contenir des circuit spécialisés pour chaque principaux **type** de donnée. Selon le **type** de la donnée, l'ensemble des opérations disponibles peut être différent.

Un langage comme le C++ est dit de «**typage fort**» car, en particulier, il y a une étape de vérification que le programme utilise les opérateurs *autorisés* pour chaque **type**.

Types élémentaires

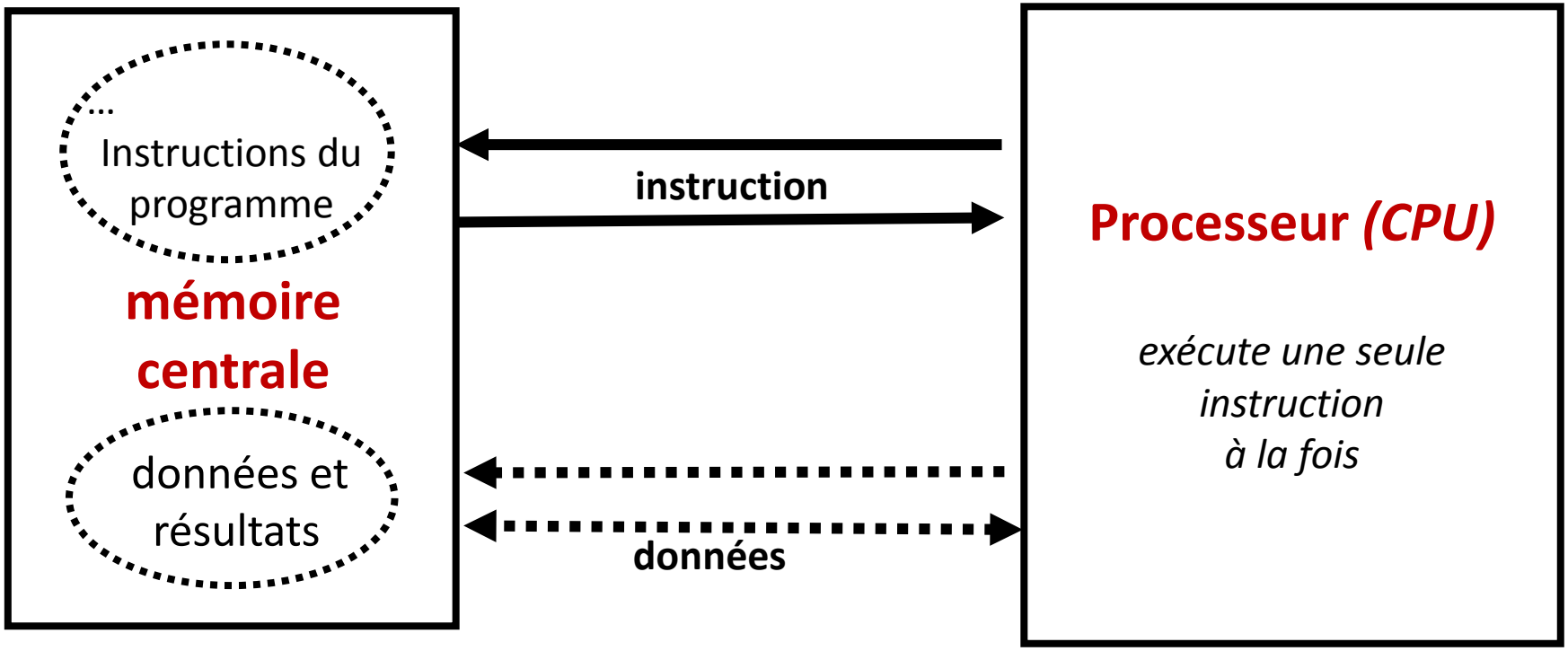
`int`	*entier signé en complément à 2*	*32 bits*
`float`	*virgule flottante IEEE 754 simple précision*	*32 bits*
`double`	*virgule flottante IEEE 754 double précision*	*64 bits*
`char`	*un seul caractère alphanumérique*	*8 bits*
`bool`	`true` *ou* `false`	

Constantes littérales

```
26
3.
'x'
```

Organisation de la mémoire centrale (1)

Les instructions ET les données à traiter ET les résultats du traitement peuvent être stockés en binaire dans un même espace: *la mémoire centrale* .



Organisation de la mémoire centrale (2)

Convention: on appelle byte un groupe de 8 bits (octet).

L'octet est la **brique de base de la mémoire centrale**

Les représentations des **données** exploitent l'octet comme élément de base.

exemples: **`char`** et **`bool`** -> 1 octet
`int` et **`float`** -> 4 octets
`double` -> 8 octets

Organisation de la mémoire centrale (3)

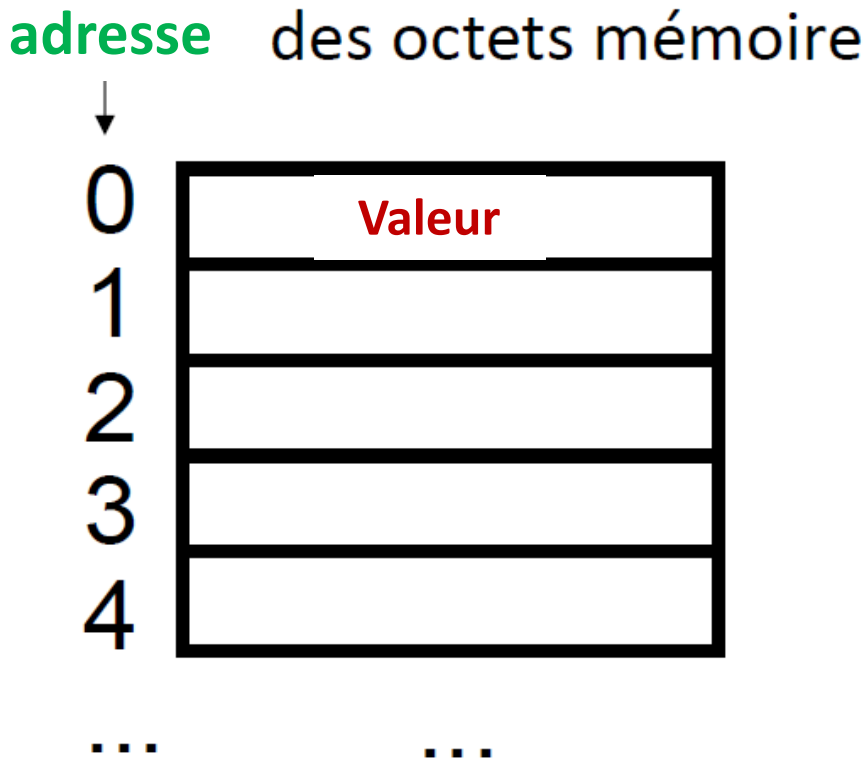
Question1: comment une instruction qui additionne deux données sait-elle où trouver les octets contenant la **valeur** de ces deux données ?

Question2: comment une instruction sait-elle où localiser les octets pour ranger une **valeur** particulière en mémoire centrale?

Réponse: chaque octet de la mémoire est numéroté ; ce numéro unique est appelé son adresse

La numérotation commence à 0

On dessine la mémoire avec l'adresse 0 en haut de la colonne d'octets



De l'octet au mot mémoire: machine 32bits, 64 bits,…

Un processeur est conçu pour que ses instructions travaillent sur un nombre prédéfini d'octets, appelé un mot :

- 4 octets : machine 32 bits
- 8 octets : machine 64 bits

La mémoire centrale est aussi organisé en mots de même taille que le processeur.

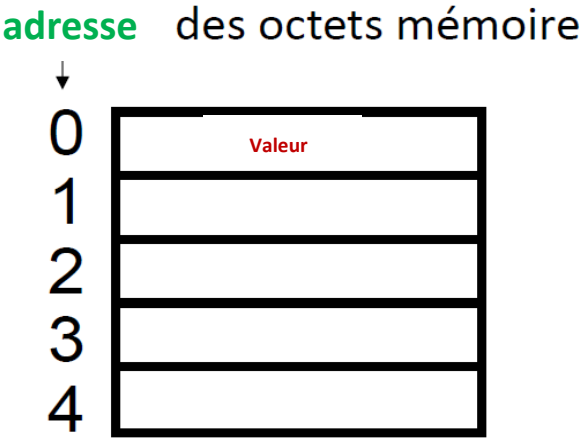
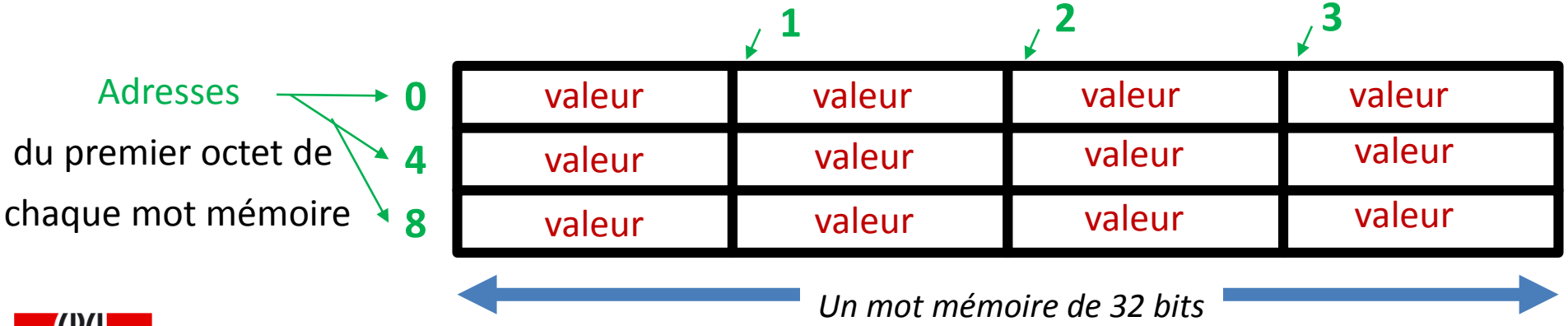


Illustration pour ***une machine 32 bits***

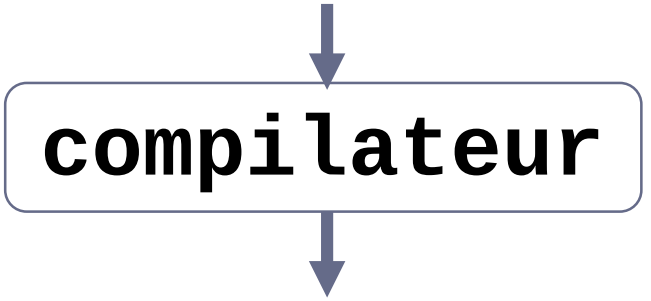


Dois-je connaître l'adresse de mes données pour coder ?

Absolument pas !

En C++ nous pouvons mémoriser la **valeur** d'une donnée d'un **type** précis dans une **variable** de ce **type** à laquelles nous donnons un **nom**, encore appelé **identificateur**.

C'est ce **nom** que nous utilisons pour désigner chaque donnée pour lui appliquer des traitements.



Le programme appelé **compilateur** est responsable de convertir le **nom** de chaque variable en une **adresse** utilisée par les instructions du processeur.

Pourquoi est-ce une bonne pratique d'initialiser une variable?

Rappels de syntaxe :

```
type nom1 ;                    (déclaration sans initialisation)
type nom2(valeur);             (déclaration avec initialisation)
type nom3 = valeur;            (déclaration avec initialisation)

nom1 = expression ;            (affectation après déclaration)
```

Exemple :

```
int x ;
++x;        // quelle est la nouvelle valeur de x ?
```

Recommandation :

```
type nom2(valeur);
```

(syntaxe **d'initialisation** sans ambiguïté)

Evaluation d'expressions avec plusieurs opérateurs

`not ! ++ --`	`* / %`	`+ -`	`< <= > >=`	`== !=`	`and &&`	`or ||`

→ Priorités décroissante des opérateurs →

Priorité identique dans chaque groupe (associativité Gauche-Droite)

Toute expression peut se réduire à une valeur possédant un type bien défini (règles de conversion)

100 – 3 + 4 * 20 / 10 % 4 + 9 – 17 % 8 * 16 % 3 - 1

Étude de cas : exercice de conversion

Conversion de degrés Fahrenheit (°F) en degrés Celsius (°C)

Ecrire un programme qui convertit une température exprimée en degrés Fahrenheit (saisie au clavier) en degrés Celsius.

La formule de conversion est: degC = (degF - 32) * (5 / 9)
où degC et degF sont les températures exprimées en degrés Celsius et Fahrenheit, respectivement.

Constantes littérales: brutes, `const`, `constexpr` ou `define` ?

La résolution d'un problème implique souvent des constantes littérales utilisées dans des calculs ou pour coder des informations.

Si une constante est un ***paramètre*** du problèmes à résoudre, il s'agit d'un ***magic number***. Il convient de *lui donner un nom* à l'aide d'une **variable** dont la valeur reste fixe grâce à **`const`** ou **`constexpr`**.

```
const double ratio_porte(largeur/2.1);
constexpr double budget(987234654,34);
```

Certaines constantes littérales telles que **M_PI** sont nommées à l'aide de la directive **`define`** et disponibles avec <cmath>, <limits>, etc…

Si par contre une constante littérale n'a aucune raison de changer (formule mathématique) on peut garder la valeur brute (ex: calcul discriminant).

Constantes littérales: codage d'information avec enum

Le mot clef **enum** sert a définir un ensemble de constantes entières

```
enum Fruit {POMME, POIRE, ABRICOT, CERISE, ORANGE};
enum Couleur {ROUGE, VERT, BLEU, JAUNE, BLANC};
```

- Le nom d'énumération (**Fruit, Couleur**) est **optionnel** ;
  il peut servir à titre documentaire
- La première valeur de constante est toujours **0**, la suivante augmente de **1**, etc...
- Il est possible d'initialiser une des constantes avec une valeur entière

```
enum Mois {JANVIER=1, FEVRIER, MARS, etc ... };
```

Cycle de développement