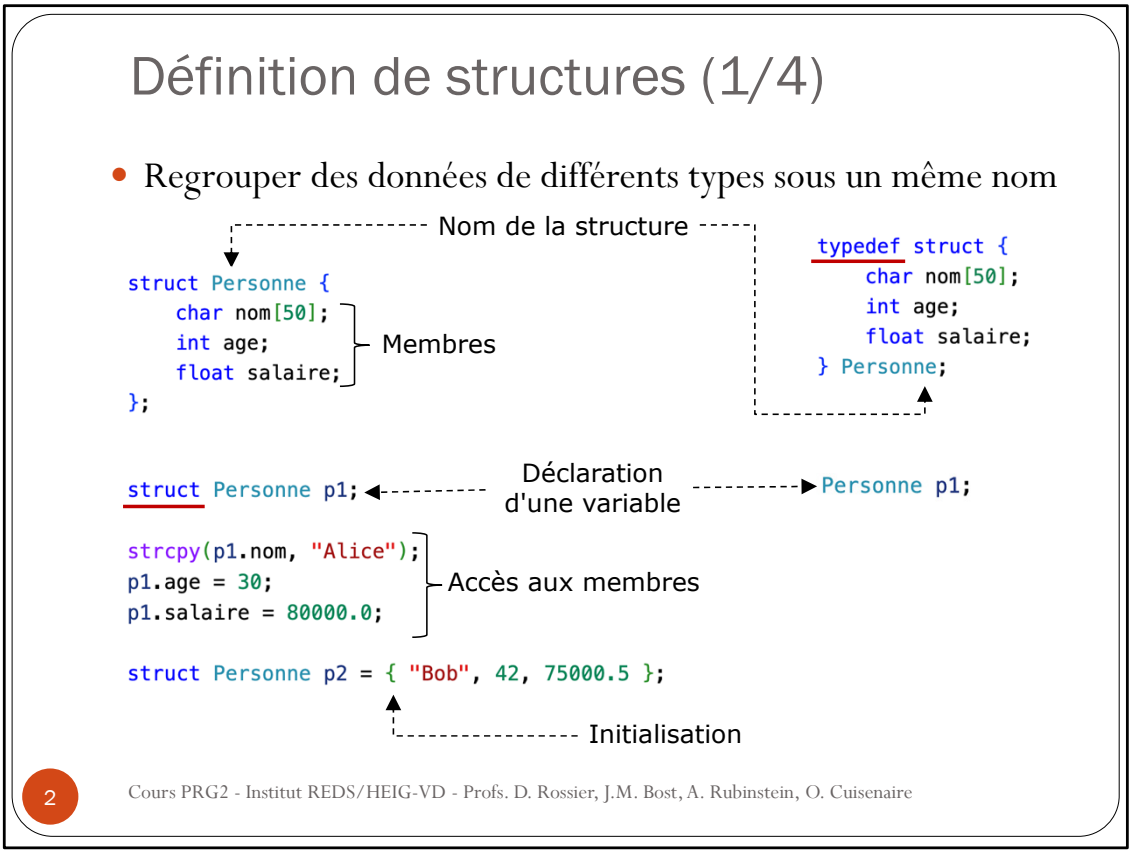
Cours PRG2

Programmation C

C7 *(2024_1)*

Contenu

- Définition de structures
- Énumérations
- Unions
- Notion d'alignement
- Utilisation de constantes



Une structure est un outil fondamental en C qui permet de regrouper plusieurs variables de types différents sous un même nom. Cela facilite la gestion de données complexes, car on peut manipuler ces différentes données comme une unité.

En C, il y a deux méthodes courantes pour déclarer une structure. La première, illustrée à gauche, utilise directement le mot-clé **struct**, suivi du nom de la structure et de ses membres. Chaque fois que vous voulez déclarer une variable de ce type de structure, vous devez préfixer la déclaration avec **struct** et le nom de la structure, comme dans **struct Personne p1;**. La seconde méthode (à droite) utilise le mot-clé **typedef**, qui crée un alias pour un type. Dans cet exemple, **typedef struct { ... } Personne;** permet de déclarer une nouvelle variable simplement avec **Personne p1;** sans avoir besoin du préfixe **struct**.

La structure **Personne** que nous avons ici est composée de trois *membres* : **nom** (un tableau de caractères), **age** (un entier), et **salaire** (un flottant). Cette organisation reflète un enregistrement individuel d'une personne dans un système. On peut accéder aux membres en utilisant la notation pointée pour affecter ou lire des valeurs, comme montré dans les instructions **p1.age = 30;** ou **strcpy(p1.nom, "Alice");**.

Enfin, l'initialisation d'une structure peut être réalisée de manière plus succincte lors de la déclaration, comme illustré avec la variable **p2**. Cette forme d'initialisation assigne directement des valeurs aux membres correspondants dans l'ordre où ils sont déclarés dans la structure. Cela rend le code plus compact lorsqu'on initialise plusieurs instances avec des valeurs connues.

Définition de structures (2/4)

- Pointeurs et Structures

```c
struct Personne *p3 = malloc(sizeof(struct Personne));

if (p3 != NULL) {
    strcpy(p3->nom, "Charlie");
    p3->age = 25;
    (*p3).salaire = 60000.0;
    p3.salaire = 25.0; <---------- Erreur de compilation
} else {
    printf("Failed to allocate memory for p3\n");
    return 1;
}

free(p3);

struct Personne *p4 = &p1;
```


Ce slide illustre l'allocation dynamique de mémoire pour une structure à l'aide du pointeur p3. En utilisant la fonction malloc, on alloue de la mémoire pour une instance de la structure Personne. Par défaut, un pointeur non initialisé pointera vers une adresse indéterminée; une déclaration de pointeur, telle que **struct Personne *p3; (**sans initialisation) ne garantit pas que **p3** soit NULL.

Dans l'exemple présenté, les membres de la structure sont assignés à travers le pointeur **p3**. Notez l'utilisation de la flèche (**->**) pour accéder aux membres de la structure via un pointeur, tandis que l'erreur montrée avec ***p3.salaire** illustre une tentative incorrecte d'accéder au membre en utilisant à la fois le déréférencement et la notation point. Enfin, free(p3); est utilisé pour libérer la mémoire allouée.

Il est aussi montré comment pointer vers une structure existante en assignant l'adresse d'une autre variable structure à un nouveau pointeur, comme avec ***p4 = &p1;**. Cela permet de travailler avec la même instance de structure sans créer une copie.

Définition de structures (3/4)

- Accès aux pointeurs dans une structure

```
struct Personne {
    char *nom;
    int age;
};

struct Personne quelquun;
struct Personne *quelquunPtr = malloc(sizeof(struct Personne));

quelquun.nom = malloc(LONGUEUR_NOM);
quelquunPtr->nom = malloc(LONGUEUR_NOM);

if (quelquun.nom != NULL && quelquunPtr->nom != NULL) {
    strcpy(quelquun.nom, "Alice");
    strcpy(quelquunPtr->nom, "Bob");
}

free(quelquun.nom);
free(quelquunPtr->nom);

free(quelquunPtr);
```

Allocation pour **la** structure

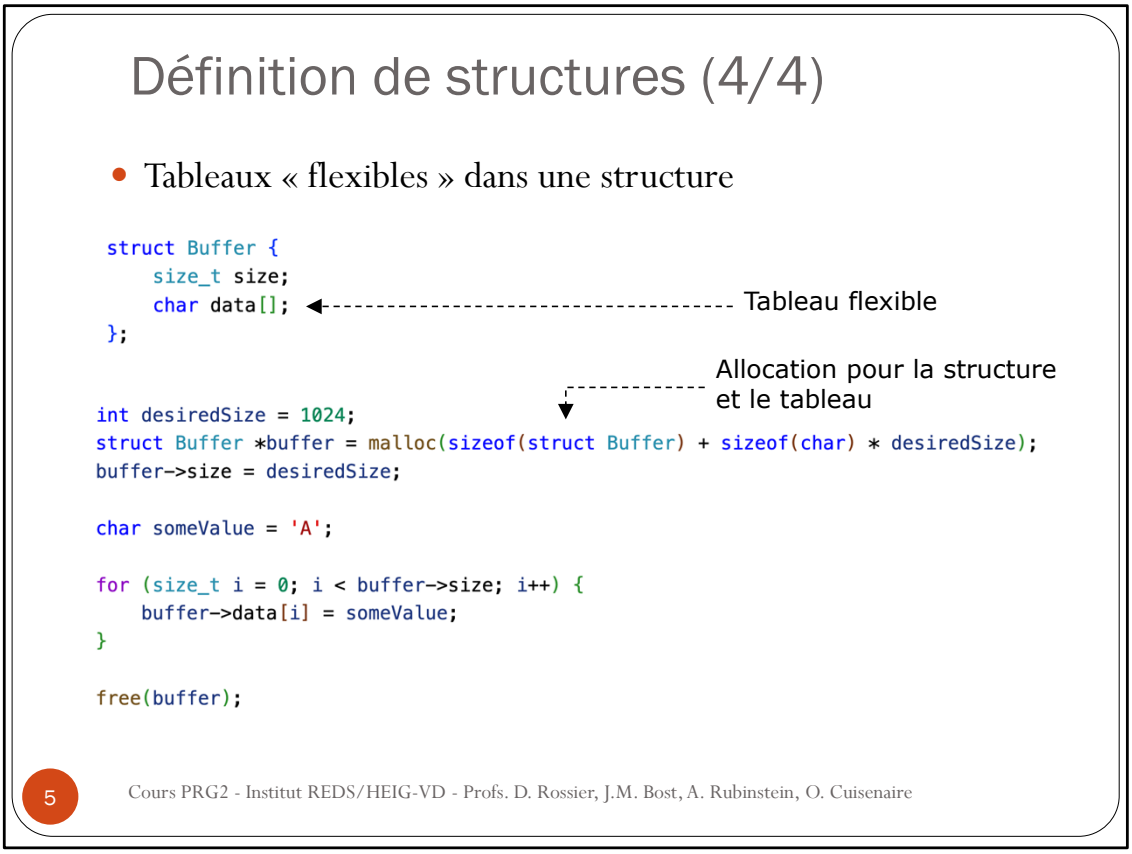
Allocation pour le pointeur **dans** la structure

Toute la mémoire doit être libérée

Lorsqu'une structure contient un ou plusieurs pointeurs comme membres, la gestion de la mémoire devient un peu plus complexe, mais aussi plus puissante. Ici, la structure **Personne** a été définie avec un membre **nom** qui est un pointeur vers **char**. Cela signifie que chaque **Personne** peut avoir un nom de longueur variable, que nous pouvons allouer dynamiquement en cours d'exécution.

Dans cet exemple, nous voyons l'allocation de mémoire pour la structure elle-même (**quelquunPtr**) et aussi séparément pour le pointeur **nom** (qui sera alloué dans le tas) à l'intérieur de la structure. Cela se fait par trois appels à `malloc()` : le premier pour créer l'instance de la structure, et les deux derniers pour allouer de la mémoire pour le nom que les structures **Personne** pointeront.

Enfin, il est essentiel de libérer toute la mémoire allouée.



Les tableaux flexibles sont une caractéristique introduite dans la norme C99 qui permet d'utiliser un tableau à taille variable directement dans une structure. Ce tableau ne doit pas avoir une taille définie et il doit être le **dernier** membre de la structure.

Dans notre exemple avec la structure **Buffer**, le tableau flexible **data** est le dernier membre de la structure. Il est crucial que le tableau flexible soit le seul tableau sans taille prédéfinie. L'allocation se fait en combinant la taille de la structure (sans le tableau flexible) et la taille du tableau désiré, calculée dynamiquement.

Comme le montre l'utilisation de `malloc()`, il faut spécifier la taille de la structure sans le tableau, puis ajouter l'espace nécessaire pour le tableau. Le tableau peut ensuite être utilisé comme n'importe quel autre tableau en C.

Enumérations (1/2)

- Permettent de définir des constantes nommées pour améliorer la lisibilité du code

```
enum Jour {
    LUNDI,     <------ 0
    MARDI,     <------ 1
    MERCREDI,  <------ 2
    JEUDI,     <------ 3
    VENDREDI,  <------ 4
    SAMEDI,    <------ 5
    DIMANCHE   <------ 6
};
```

```
enum Couleurs {
    ROUGE = 1, <------ 1
    VERT,      <------ 2
    BLEU       <------ 3
};
```

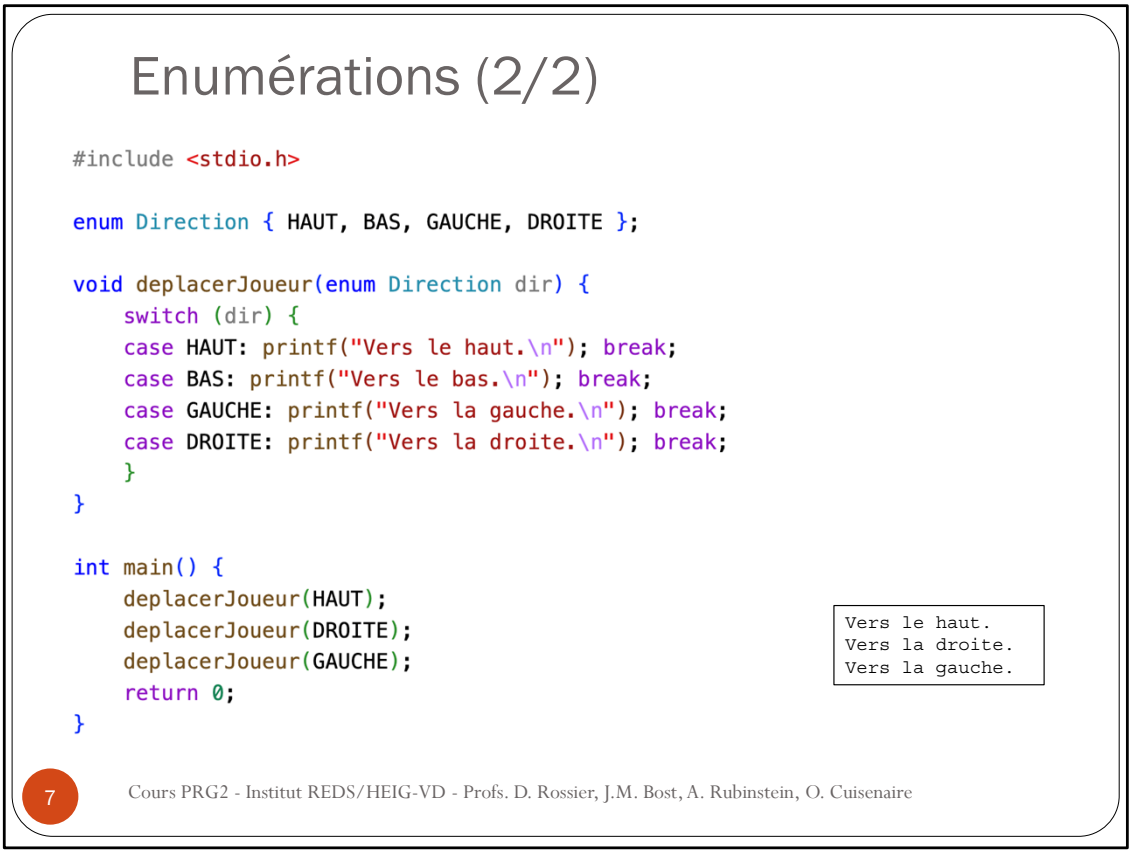
```
enum Temperature {
    FROID = -10,    <------ -10
    GLACIAL = -20,  <------ -20
    CHAUD = 30,     <------  30
    BRULANT = 50    <------  50
};
```


Les énumérations sont une fonctionnalité du langage C qui permettent de créer un type de données composé de constantes nommées. L'usage d'énumérations améliore la lisibilité du code en remplaçant les *nombres magiques*, qui sont souvent source de confusion, par des noms significatifs et explicites. Cela rend le code plus maintenable et plus facile à comprendre.

L'énumération **Jour**, par exemple, définit une suite de constantes qui représentent les jours de la semaine. En C, si la première valeur n'est pas explicitement initialisée, elle est par défaut mise à 0, et chaque nom suivant dans l'énumération est incrémenté de 1. Cela signifie que **LUNDI** vaudra 0, **MARDI** vaudra 1, et ainsi de suite.

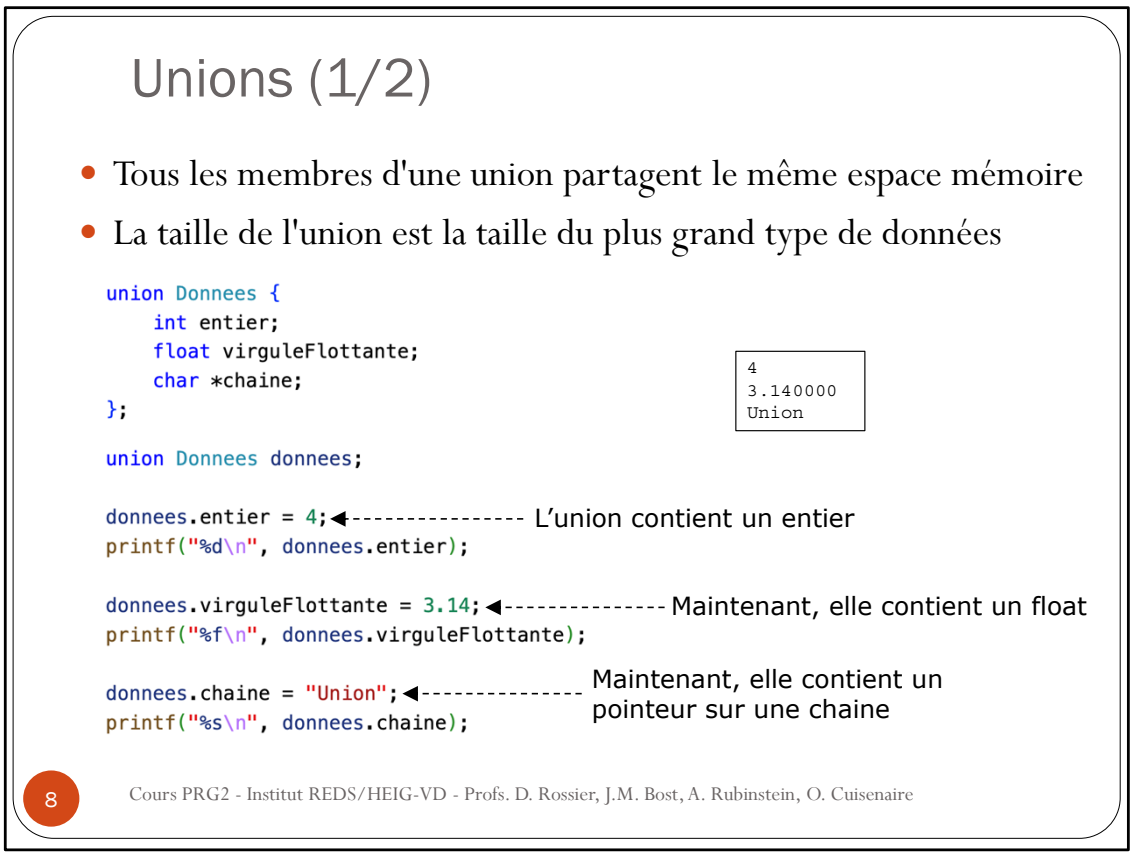
Dans l'énumération **Couleurs**, nous initialisons explicitement **ROUGE** à 1. Par conséquent, les valeurs suivantes, **VERT** et **BLEU**, seront automatiquement assignées aux entiers 2 et 3, respectivement.

Enfin, avec l'énumération **Temperature**, nous montrons que les énumérations ne sont pas limitées à des séquences ordonnées. Ici, des températures spécifiques sont assignées à des descripteurs tels que **FROID**, **GLACIAL**, **CHAUD**, et **BRULANT**, avec des valeurs arbitraires pouvant aussi être négatives.



Cet exemple de code illustre une utilisation pratique des énumérations en C : la définition de directions pour le mouvement d'un joueur dans un jeu ou une application. L'énumération **Direction** est définie avec quatre directions possibles : **HAUT**, **BAS**, **GAUCHE**, **DROITE**. Ces constantes nommées rendent les appels de fonction et les contrôles de la logique beaucoup plus clairs que si des valeurs numériques étaient utilisées.

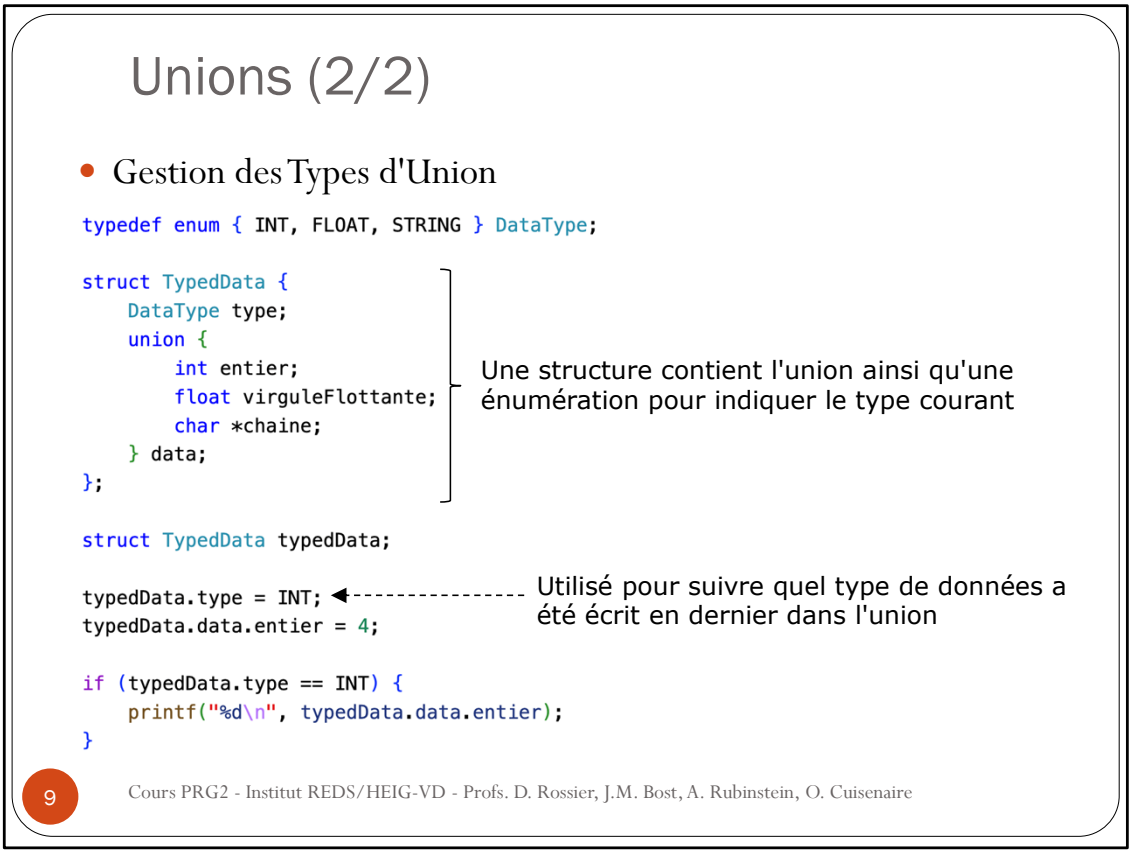
La fonction `deplacerJoueur()` prend une direction en paramètre et utilise une instruction ***switch*** pour exécuter un bloc de code différent selon la direction donnée. Cela montre comment les énumérations peuvent simplifier le contrôle de flux avec des conditions multiples basées sur un ensemble de valeurs prédéterminées. Plutôt que de se rappeler des nombres arbitraires pour chaque direction, le programmeur peut utiliser les noms des énumérations, ce qui rend le code plus facile à lire et à maintenir.



Les unions permettent de stocker différents types de données dans le même espace mémoire. Elles sont utiles quand vous avez besoin de travailler avec des données qui peuvent prendre plusieurs formes mais pas simultanément. Contrairement aux structures, où chaque membre possède sa propre zone de mémoire, une union alloue un unique bloc de mémoire suffisamment grand pour stocker le plus grand de ses membres, de sorte que l'union dans cet exemple aura la taille d'un float ou d'un pointeur sur char, selon ce qui est le plus grand sur la plateforme cible.

Dans l'exemple fourni, l'union **Donnees** peut contenir soit un **int**, soit un **float**, soit un pointeur de **char**. Quand une valeur est assignée à un membre, l'espace mémoire de l'union est utilisé pour cette valeur. Si plus tard une autre valeur est assignée à un autre membre, elle écrase la valeur précédente, car les membres partagent la même localisation en mémoire.

L'utilisation de **donnes.entier = 4;** assigne la valeur 4 à la partie de l'union qui traite les entiers. Ensuite, **donnes.virguleFlottante = 3.14;** réassigne la même zone de mémoire pour stocker la valeur flottante. De même, **donnes.chaine = "Union";** utilise l'espace pour stocker un pointeur vers une chaîne de caractères. Il est important de ne pas oublier que seul le dernier membre assigné contient une donnée valide, car chaque assignation écrase les précédentes.



Cet exemple illustre comment une structure peut servir de capsule pour une union, permettant une gestion sécurisée et flexible des données hétérogènes.

La structure **TypedData** combine une union avec une énumération **DataType** qui sert de marqueur pour le type de données actuellement stockées dans l'union. Cette méthode est communément appelée "union typée" ou "variant".

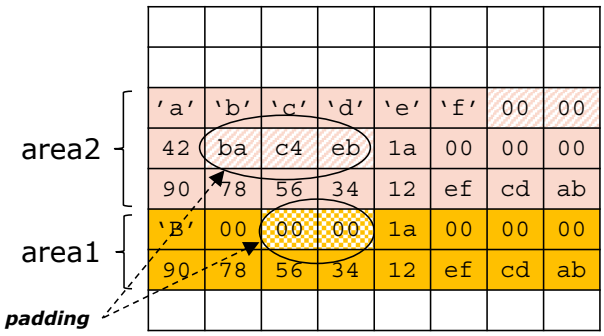
L'énumération **DataType** définit un ensemble de types possibles qui peuvent être stockés dans l'union : **INT**, **FLOAT**, et **STRING**. Lorsque des données sont assignées à l'union, le type correspondant est aussi mis à jour dans la variable type. Ceci permet de savoir quel membre de l'union contient des données valides à un moment donné.

Dans l'exemple de code, **typedData.type = INT;** indique que l'union va stocker un entier. On peut donc vérifier le type avant d'utiliser la valeur avec un **if**, par exemple. Cette vérification garantit que le bon membre de l'union est utilisé pour éviter d'interpréter incorrectement les données.

Notion d'alignement (1/3)

- Alignement des données en mémoire

```
typedef struct {
    char *a;
    char b;
    int c;
} st1;

typedef struct {
    char *a;
    char b;
    int c;
    char d[6];
} st2;


int main() {

  st1 area1 = { (void *) 0xabcdef1234567890, 'B', 26 };
  st2 area2 = { (void *) 0xabcdef1234567890, 'B', 26, { 'a', 'b', 'c', 'd', 'e', 'f' } };

  printf("sizeof st1 = %zu\n", sizeof(st1));
  printf("sizeof st2 = %zu\n", sizeof(st2));

  exit(EXIT_SUCCESS);
}
```




```
sizeof st1 = 16
sizeof st2 = 24
```


L'**alignement des données en mémoire** est un concept fondamental en programmation.

Lorsque nous stockons des données dans la mémoire d'un ordinateur, elles sont organisées en octets. L'alignement des données se réfère à la manière dont ces octets sont disposés. Cette disposition peut affecter la performance car les processeurs sont optimisés pour accéder aux données alignées plus rapidement.

Différents types de données ont des exigences spécifiques en matière d'alignement.
Par exemple, les entiers doivent souvent être alignés sur des adresses multiples de 4 ou 8 octets. Lorsque nous utilisons des structures (comme «*struct*» en C), l'alignement des champs est crucial.

Les compilateurs insèrent parfois un "rembourrage" (*padding*) pour aligner les champs correctement.

Notion d'alignement (2/3)

- Compaction des données
- Accès mémoire avec un mappage de structure

```
typedef struct __attribute__((packed)) {
    char *a;
    char b;
    int c;
} st1;

typedef struct __attribute__((packed)) {
    char *a;
    char b;
    int c;
    char d[6];
} st2;
```

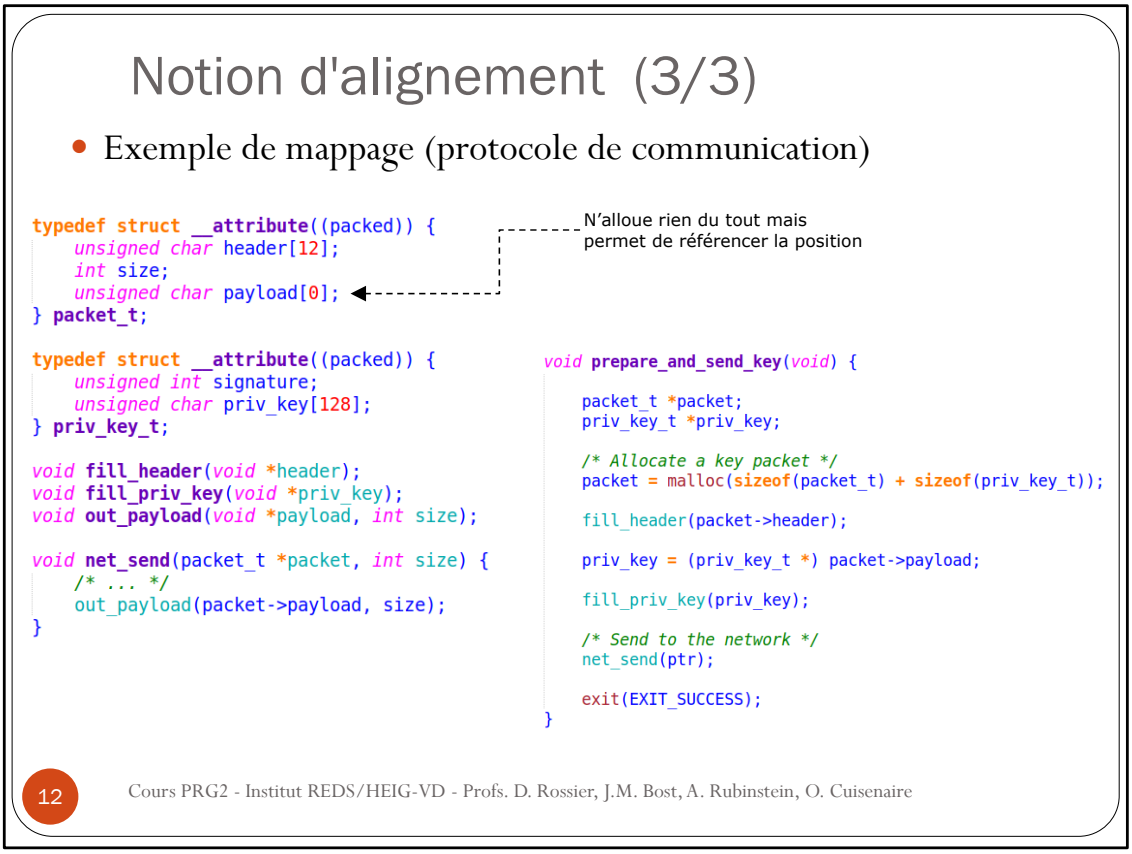
Attribut *gcc*

```
sizeof st1 = 13
sizeof st2 = 19
```

00	00	'a'	'b'	'c'	'd'	'e'	'f'
34	12	ef	cd	ab	42	1a	00
'B'	1a	00	00	00	90	78	56
90	78	56	34	12	ef	cd	ab

L'alignement par défaut peut entraîner un gaspillage d'espace mémoire et rendre difficile le mappage de structure sur des zones mémoires. Avec le compilateur *gcc* par exemple, il est possible de **forcer un alignement compact** en utilisant des attributs spécifiques, tels que `__attribute__((packed))` comme l'illustre l'exemple ci-dessus.

Ainsi la déclaration des structures entraînent une réduction de la place allouée et permette d'accéder les données de manière contiguë.



L'exemple présenté illustre l'utilisation de deux structures dans le contexte d'un protocole de communication.

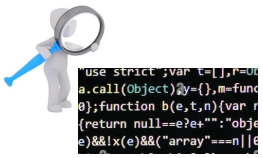
La structure `packet_t` définit le contenu d'un paquet, composé d'un en-tête et d'une charge utile de taille spécifique. Le champ de la charge utile (***payload***) est déclaré comme un tableau de type "**zéro élément**", ce qui signifie qu'aucune allocation mémoire n'est effectuée pour ce champ ; cela permet de référencer efficacement la position du ***payload***, qui sera accolé à la structure d'en-tête.

La structure contenue dans le *payload* est définie pour représenter une clé privée. L'utilisation d'une allocation dynamique assure la contiguïté des données à l'intérieur du paquet.

La fonction `net_send()` peut aisément se référer au *payload* grâce à l'utilisation du champ *payload*.

Utilisation de constantes (1/5)

- **Immuabilité** et **sécurité**
  - Définition de valeurs **immuables**
    - Assure **l'intégrité** des données.
  - Les pointeurs constants empêchent la **modification de l'adresse pointée**.



- **Clarté du code**
  - Utilisation de constantes pour des pointeurs afin de rendre le code plus lisible et compréhensible.

- **Utilisation courante**
  - Utilisation de constantes avec des pointeurs pour **passer des données de manière sécurisée dans les fonctions**.

Les **constantes** sont des valeurs qui restent inchangées pendant l'exécution d'un programme. Elles jouent un rôle essentiel dans la programmation, car elles permettent de fixer des valeurs **immuables**.

L'utilisation de constantes améliore la lisibilité du code et favorise la maintenance.

Utilisation de constantes (2/5)

- 4 types de constante
  - **Entier**
    - **Décimal**
    - **Octal**
    - **Hexadécimal**
  - **Flottant (nombre réel)**
  - **Caractère** (ou chaîne de caractères)
    - `'X', '\33', '\x1b', '\033', '\n',` etc.
    - "Hello World", "Une plus longe \ chaîne de caractères"
  - **Enumération**

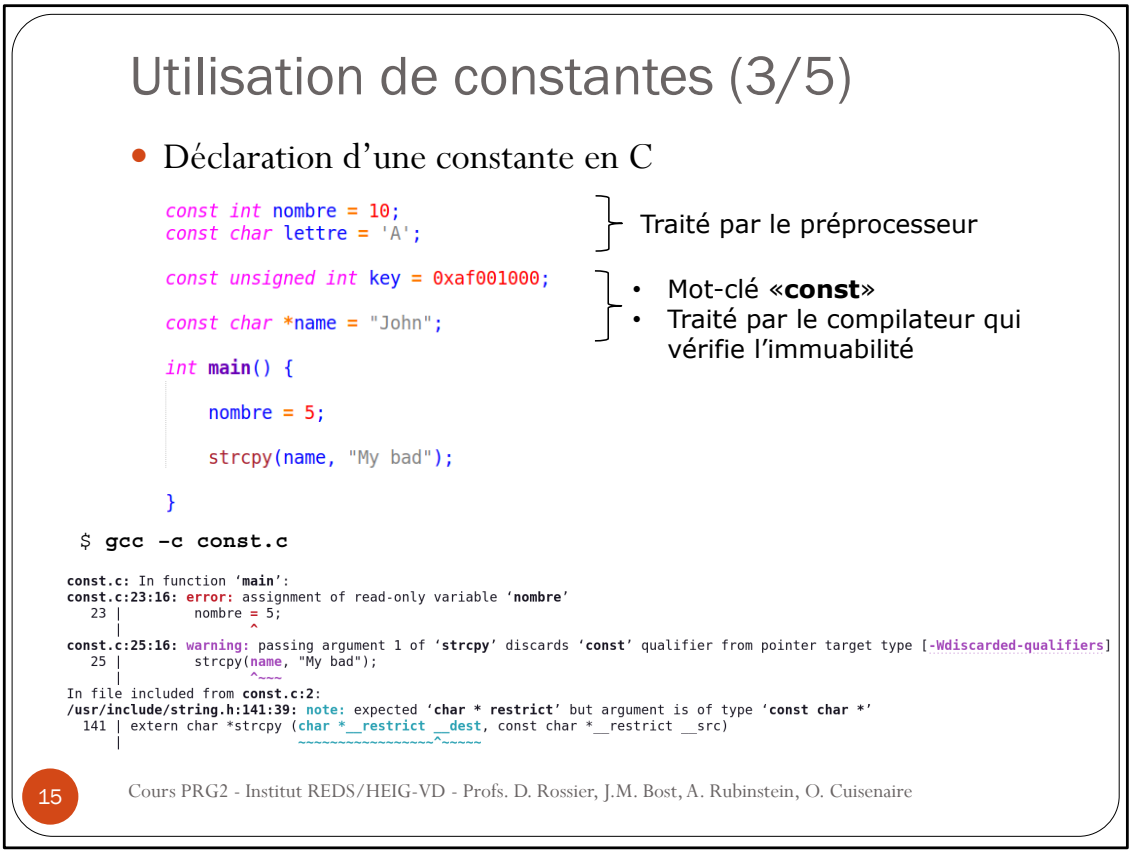
Constante	Type
`1234`	`int`
`02322`	`int /* octal */`
`0xa4d2`	`int /* hexadécimal */`
`123456789L`	`long`
`1234U`	`unsigned int`
`123456789UL`	`unsigned long int`

Constante	Type
`12.34`	`double`
`12.3e-4`	`double`
`12.34F`	`float`
`12.34L`	`long double`

Une constante est un littéral portant un nom. Selon le type de la constante, le littéral peut être un nombre (réel ou flottant), un caractère ou une chaîne de caractères ou une énumération.

Avec un littéral numérique, le préfixe `0` (zèro) signifie que le nombre est représenté dans la base 8 (octale) alors que le préfixe `0x` est utilisé pour la représentation des nombres dans la base 16 (hexadécimale). De plus, un suffixe peut être utilisé afin de préciser le type de la valeur. Par exemple, le suffixe `UL` signifie que le nombre doit être considéré comme un entier long non-signé, alors que `LL` signifie un nombre de type long long (signé).

Les préfixes et suffixes peuvent être exprimés indifféremment en majuscule ou en minuscule.



Il faut bien faire la différence entre une constante exprimée à l'aide de `#define` et celle avec `const`. Comme on l'a vu précédemment, la directive `#define` est une directive de préprocesseur liée à la notion de ***macro***, et où chaque occurrence de la macro est remplacée par sa valeur lors de la compilation (préprocesseur). Par conséquent, elle ne permet pas de définir une variable.

En revanche, l'utilisation de const qualifie une variable comme étant en lecture seule (la variable est alors dite ***immuable***). La directive `const` crée une véritable variable avec un type et une adresse mémoire (on peut pointer vers une variable `const`).

Le compilateur peut ainsi vérifier la compatibilité des types et signaler une erreur lors d'une tentative d'écriture (modification) d'une variable `const`.

Utilisation de constantes (4/5)

- Constantes et pointeurs

```
int x = 10;
const int *ptr = &x;
int * const ptr2 = &x;
const int * const ptr3 = &x;
int main() {
    int y;
    *ptr = 100;
    ptr = &y;
    *ptr = 200;
    ptr2 = &y;
    ptr3 = ptr2;
}
```

◄------- La **valeur pointée** est immuable

◄------- La **valeur du pointeur** est immuable

◄--------- **Valeur pointée** et **valeur du pointeur** immuables

```
$ gcc -c const.c

const.c: In function 'main':
const.c:28:14: error: assignment of read-only location '*ptr'
   28 |         *ptr = 100;
      |         ^
const.c:30:14: error: assignment of read-only location '*ptr'
   30 |         *ptr = 200;
      |         ^
const.c:32:14: error: assignment of read-only variable 'ptr2'
   32 |         ptr2 = &y;
      |         ^
const.c:34:10: error: assignment of read-only variable 'ptr3'
   34 |     ptr3 = ptr2;
      |          ^
```


L'utilisation de `const` avec les pointeurs est très courante car elle permet de protéger le contenu référé par le pointeur ainsi que le pointeur lui-même. Pour cela, il existe trois formes possibles :

- `const` s'applique sur le contenu pointé
  - Exemple : `const char *ptr;` (`const` avant le type)
- `const` s'applique sur le pointeur lui-même.
  - Exemple : `char * const ptr;` (ou `char *const ptr`)
- `const` s'applique sur le pointeur lui-même **et** le contenu pointé.
  - Exemple : `const char * const ptr;`

Le compilateur est en mesure de générer un `warning` ou une erreur en cas d'une tentative de modification du pointeur et/ou du contenu.

Utilisation de constantes (5/5)

- Exemples de conversions

```c
int main() {
    int num = 20;
    const int *ptr1 = &num; // Pointeur constant vers une variable non constante
    int *ptr2;

    const int value = 100;
    const int * const constPtr = &value; // Pointeur constant vers une valeur constante

    // Conversion d'un pointeur constant à non constant
    ptr2 = (int *) ptr1; // Conversion explicite du type de pointeur

    // Modification de la valeur pointée par le pointeur non constant
    *ptr2 = 30;

    // Conversion d'un pointeur constant const int * const à un pointeur non constant int *
    int *mutablePtr = (int *) constPtr; // Conversion explicite du type de pointeur

    // Modification de la valeur pointée par le pointeur non constant
    *mutablePtr = 200;

    return 0;
}
```


L'exemple ci-dessus montre qu'il est possible de convertir une variable constante vers une variable non constante. Bien entendu, cette dernière ne sera pas déclarée avec `const`.

La plupart des fonctions de la librairie «*string.h*» ont des arguments constants indiquant l'immuabilité des chaînes de caractères si celles-ci ne sont pas modifiées ; une chaîne de caractères n'étant pas déclarées comme constante peut sans autre être passées comme argument constant à l'une de ces fonctions.

Exemple :

```c
char *strcpy(char *dst, const char *src)

int main(int argc, char **argv) {

    char buf[80];

    strcpy(buf, argv[0]);

}
```