Types composés

ICC-C Cours 10

Data

- Nous voulons traiter des données tabulaires
- Par exemple, des données personnelles, ou les caractéristiques d'un produit
- Souvent il y a plusieurs entrées qu'on veut manipuler

Nom	Prénom	Année	Mois	Jour	Taille (cm)	Poids (kg)
Meunier	Alice	2002	11	21	168	55.3
Smith	Robert	1977	5	2	179	81.2

Comment ferions nous?

- On peut définir des tableaux qui stockent chacun une colonne

```
char *nom[100], *prénom[100];
int année[100], mois[100], jour[100];
int taille_cm[100];
float poids_kg[100];
```

- Pour accéder aux infos d'une personne, on utilise partout le même indice
- Stockage par colonne = *Column store*
- Il y a effectivement des systèmes qui stockent les données de cette manière!

Et le stockage par ligne?

- Nous avons besoin d'un nouveau type composé!
- Les tableaux stockent plusieurs valeurs du même type
- Une structure stocke des valeurs de plusieurs types
- On définit le type `struct` ainsi:

```
struct
{
    type_1 membre_1;
    type_2 membre_2;
    ...
    type_n membre_n;
}
```

Les champs du struct s'appellent des membres

```
struct
{
    char *nom;
    char *prénom;
    int année;
    int mois;
    int jour;
    int taille_cm;
    float poids_kg;
}
```

Une variable de type `struct`

- C’est un peu long à écrire…
- …mais la variable `bob` peut stocker les champs dont on a besoin
- Heureusement, `struct` peut avoir un nom!

```
struct
{
    char *nom;
    char *prénom;
    int année;
    int mois;
    int jour;
    int taille_cm;
    float poids_kg;
} bob;
```

Une structure nommée

Named struct

- Dans le code on peut définir une structure avec un nom

```
struct personal_info
{
    char *nom;
    char *prénom;
    int année;
    int mois;
    int jour;
    int taille_cm;
    float poids_kg;
};
```

Définition du type
`struct personal_info`

- La déclaration d'une variable est beaucoup plus brève:

```
struct personal_info bob;
```

Variable de type
`struct personal_info`

Encore plus bref?

- C’est possible avec `typedef`
- Syntaxe:
  `typedef <type> <alias>;`
- `Exemples:`

```
typedef int *pointeur_int;
typedef char string99[100]; // Pour les tableaux le [] vient après l’alias
typedef float petit_réel;
typedef double gros_réel;
typedef char **double_pointeur_char;

pointeur_int ptr;
string99 coucou = "coucou";
```

Encore plus bref?

```
typedef struct personal_info
{
    char *nom;
    char *prénom;
    int année;
    int mois;
    int jour;
    int taille_cm;
    float poids_kg;
} personal_info_t;
```

- Pour notre exemple, le type `personal_info_t` est synonyme de la structure nommée
- Pour définir une variable on peut écrire:

```
personal_info_t bob;
```

Initialisation & affectation

- Comme dans le cas des tableaux, on peut initialiser une variable `struct` lors de sa définition:

```
personal_info_t bob = {"Smith", "Robert", 1977, 5, 2, 179, 81.2};
```

- On peut aussi affecter le contenu d'une variable `struct` à une autre variable:

```
personal_info_t bob_copie;

bob_copie = bob;
```

- Rappel: on ne pouvait pas faire ça avec des tableaux!

Accéder aux membres

```
personal_info_t bob = {"Smith", "Robert", 1977, 5, 2, 179, 81.2};
```

- Comment accède-t-on aux infos de Bob?
- On utilise l'opérateur d'accès aux membres "."

```
printf("%s %s pèse %.2f kg\n",
       bob.prénom,
       bob.nom,
       bob.poids_kg);
// Affiche: Robert Smith pèse 81.2 kg
```

```
typedef struct personal_info
{
    char *nom;
    char *prénom;
    int année;
    int mois;
    int jour;
    int taille_cm;
    float poids_kg;
} personal_info_t;
```

Accéder aux membres

Pointeurs

- Parfois on utilise des pointeurs vers une variable `struct`

```
personal_info_t bob = {"Smith", "Robert", 1977, 5, 2, 179, 81.2};

personal_info_t *p_info = &bob;
```

- Normalement on devrait utiliser l'opérateur d'indiréction:

```
(*p_info).taille_cm;
```

- Mais il y a plus intuitif avec l'opérateur "–>" (c'est équivalent)

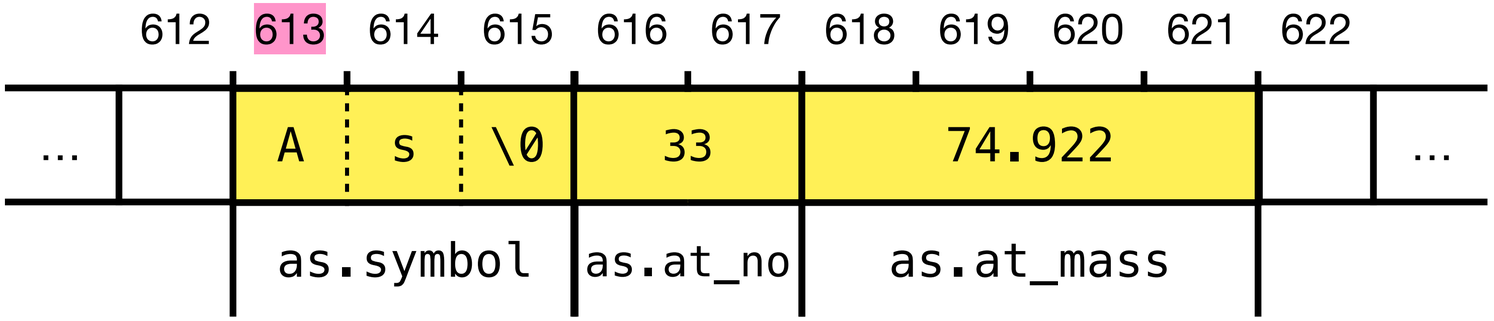
```
printf("%s %s mesure %d cm\n",
       p_info->prénom,
       p_info->nom,
       p_info->taille_cm);
// Affiche: Robert Smith mesure 179 cm
```

Comment est-ce stocké en mémoire?

- L’ordre des champs compte
- Un élément chimique a
  - un symbole d’au plus 2 chars
  - Numéro atomique (petit entier)
  - Masse atomique (réel)

```
typedef struct chem_element
{
    char symbol[3];
    short at_no;
    float at_mass;
} chem_element_t;

chem_element_t as =
 {"As", 33, 74.922};
```



Passage par valeur

- Que se passe-t-il quand on passe une structure en argument d'une fonction?
- La fonction `sneaky` essaye de modifier le numéro atomique
- Qu'adviendra-t-il ? 😬

```c
typedef struct chem_element
{
  char symbol[3];
  short at_no;
  float at_mass;
} chem_element_t;

void sneaky(chem_element_t e)
{
  e.at_no++;
}

int main()
{
  chem_element_t as =
        {"As", 33, 74.922};
  printf("No atomique = %d\n”,
         as.at_no);
  sneaky(as);
  printf("No atomique = %d\n”,
         as.at_no);
}
```

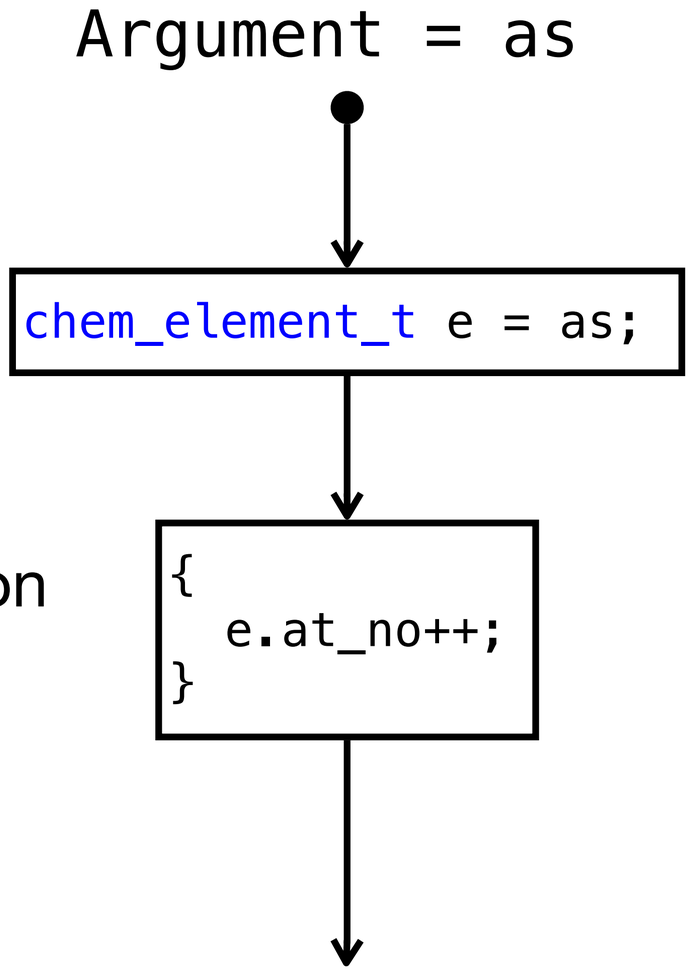
Appel de fonction

```
void sneaky(chem_element_t e)
{
  e.at_no++;
}
```

1. passage par valeur

2. “Le corps de la fonction est exécuté”

```
sneaky(as)
```



Puisqu’on peut affecter une struct à une autre variable, `sneaky` a modifié la copie! 😮‍💨

Tableaux de struct

```c
typedef struct _coord
{
    float x, y;
} coord_t;

coord_t triangle[] = {
  {0, 0}, {0, 1}, {1, 0}};

coord_t *hexagone = polygone(6);
```

```c
coord_t *polygone(int n)
{
    coord_t *sommets =
        malloc(n * sizeof(coord_t));

    for (int k=0; k<n; k++)
    {
        sommets[k].x = cos(2*k*M_PI/n);
        sommets[k].y = sin(2*k*M_PI/n);
    }

    return sommets;
}
```

Test d’égalité

- On ne peut pas directement comparer deux struct 😢
- On doit les comparer membre par membre…

Enumérations

- Parfois nous voulons définir des constantes qui correspondent à une énumération
- Exemples:
  - Les jours de la semaine,
  - Les sept nains,
  - Les actions possibles dans Candy Crush (too soon?)
- Il existe un type pour ça!

Enumérations

- Syntaxe:

```
enum <nom>
{
    constante_1,
    constante_2,
    ...
    constante_n
}
```

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

```
enum nain {
    Atchoum = 5,
    Dormeur,   // 6
    Grincheux, // 7
    Joyeux = 3,
    Prof,      // 4
    Simplet,   // 5
    Timide     // 6
};
```

- Les constantes (entières) auront des valeurs consécutives qui commencent à 0
- On peut modifier les valeurs des constantes, les valeurs suivantes seront aussi consécutives (pas forcément distinctes!)

Enumérations

Variables

```
jour_t j = MARDI;
```

- On peut lui affecter une autre valeur que la valeur de `MARDI`
- Les enum sont essentiellement des entiers

```
printf("Enum jour = %d (taille=%ld) \n”,
        j, sizeof(j));
// Affiche: Enum jour = 1 (taille=4)
```

- (Sauf en C23 où on peut spécifier un autre type)

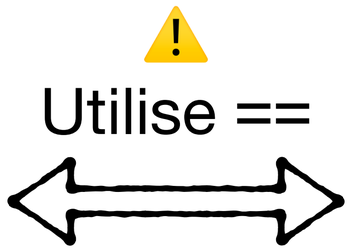
```
typedef enum _jour {
    LUNDI,
    MARDI,
    MERCREDI,
    JEUDI,
    VENDREDI,
    SAMEDI,
    DIMANCHE
} jour_t;
```

L’instruction `switch`

- Parfois nous voulons tester un nombre **discret** de valeurs
- Pour chaque valeur — exécuter un code différent

```
switch (<expression>)
{
case valeur_1:
    <instruction_1>
    break;
...
case valeur_k:
    <instruction_k>
    break;

default:
    <instruction_par_défaut>
}
```




```
T e = <expression>;

if (e == valeur_1)
  <instruction_1>
else if (e == valeur_2)
...
else if (e == valeur_k)
  <instruction_k>
else
  <instruction_par_défaut>
```

L’instruction `switch`

```
switch (jour)
{
case LUNDI:
  printf("Les canards vont à la mare.\n");
  break;
case MARDI:
  printf("Ils s'en vont jusqu'à la mer.\n");
  break;
case MERCREDI:
  printf("Ils organisent un grand jeu.\n");
  break;
case JEUDI:
  printf("Ils se promènent dans le vent.\n");
  break;
case VENDREDI:
  printf("Ils se dandinent comme ça.\n");
  break;
case SAMEDI:
  printf("Ils se lavent à ce qu'on dit.\n");
  break;
case DIMANCHE:
  printf("Ils se reposent et "
         "voient la vie en rose\n"
         "La semaine recommencera demain, "
         “coin, coin.\n");
  break;
}
```

L’instruction `switch`

Le break

```c
switch (zero_ou_un)
{
case 0:
  printf("Zéro\n");

case 1:
  printf("Un\n");
}
```

⚠

Quand le `break` manque, on passe à travers au `case` suivant!

```
zero_ou_un: 1
Affiche:
Un

zero_ou_un: 0
Affiche:
Zéro
Un
```

Etude sur le sommeil

- Chaque participant a un identifiant (entier)
- On mesure la durée du sommeil pendant un certain nombre de nuits
- Les mesures = un tableau indiquant le nombre d'heures de sommeil + jour de la semaine

```c
typedef struct _sommeil
{
    jour_t jour;
    int heures_dormies;
} sommeil_t;

typedef struct _info
{
    int identifiant;
    int nombre_nuits;
    sommeil_t *nuits;
} info_t;
```

Initialiser les mesures

Bob

```
sommeil_t sommeil_bob[] =
{
  {MARDI, 8},
  {MERCREDI, 7},
  {JEUDI, 8}
};
// Bob a dormi 8h mardi, 7h mercredi,
// et de nouveau 8h jeudi

info_t info_bob = {1, 3, sommeil_bob};
// L'identifiant de Bob est 1,
// on a 3 mesures
```

Jour	H
Mardi	8h
Mercredi	7h
Jeudi	8h

```
typedef struct _sommeil
{
    jour_t jour;
    int heures_dormies;
} sommeil_t;

typedef struct _info
{
    int identifiant;
    int nombre_nuits;
    sommeil_t *nuits;
} info_t;
```

Initialiser les mesures

Alice

Jour	H
Mardi	6h
Mercredi	10h

```
info_t info_alice = info_bob;
// On prend les valeurs de Bob et on les modifie

info_alice.identifiant = 2;
info_alice.nombre_nuits = 2;
// Que 2 nuits

info_alice.nuits[0].heures_sommeil = 6;
// Alice a dormi 6h mardi

info_alice.nuits[1].heures_sommeil = 10;
// Alice a dormi 10h mercredi

// Pas de mesure pour jeudi
```

```
typedef struct _sommeil
{
    jour_t jour;
    int heures_dormies;
} sommeil_t;

typedef struct _info
{
    int identifiant;
    int nombre_nuits;
    sommeil_t *nuits;
} info_t;
```

Que voit-on?

Bob

Jour	H
Mardi	8h
Mercredi	7h
Jeudi	8h

Alice

Jour	H
Mardi	6h
Mercredi	10h

```
Identifiant: 2
Nombre de nuits: 2
Nuit 1: Mardi, 6 heures
Nuit 2: Mercredi, 10 heures

Identifiant: 1
Nombre de nuits: 3
Nuit 1: Mardi, 6 heures
Nuit 2: Mercredi, 10 heures
Nuit 3: Jeudi, 8 heures
```

- En modifiant les mesures de Alice, nous avons aussi modifié les mesures de Bob!

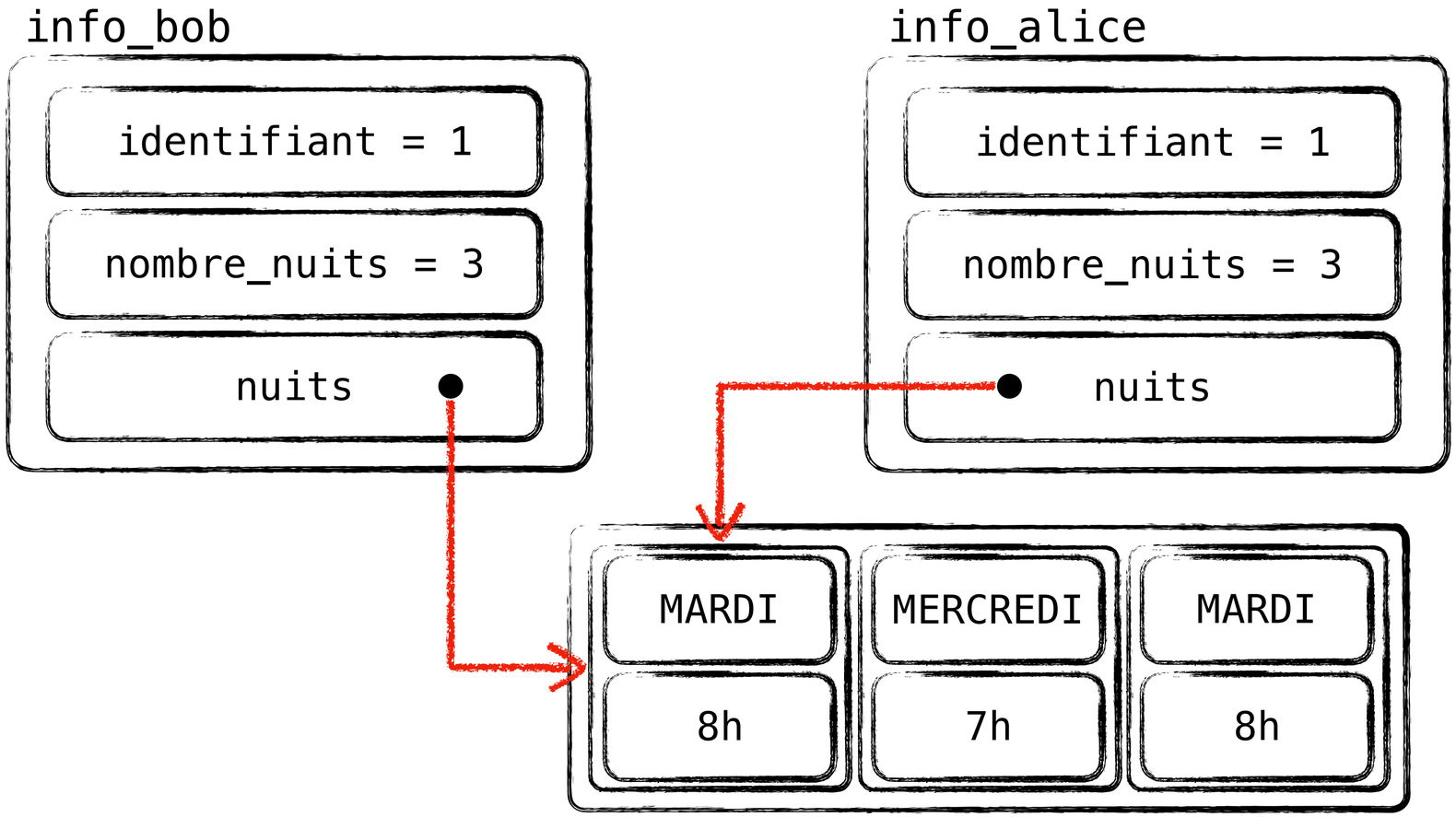
```c
typedef struct _sommeil
{
    jour_t jour;
    int heures_dormies;
} sommeil_t;

typedef struct _info
{
    int identifiant;
    int nombre_nuits;
    sommeil_t *nuits;
} info_t;
```

Copie superficielle

Shallow copy

```
info_t info_alice = info_bob;
```



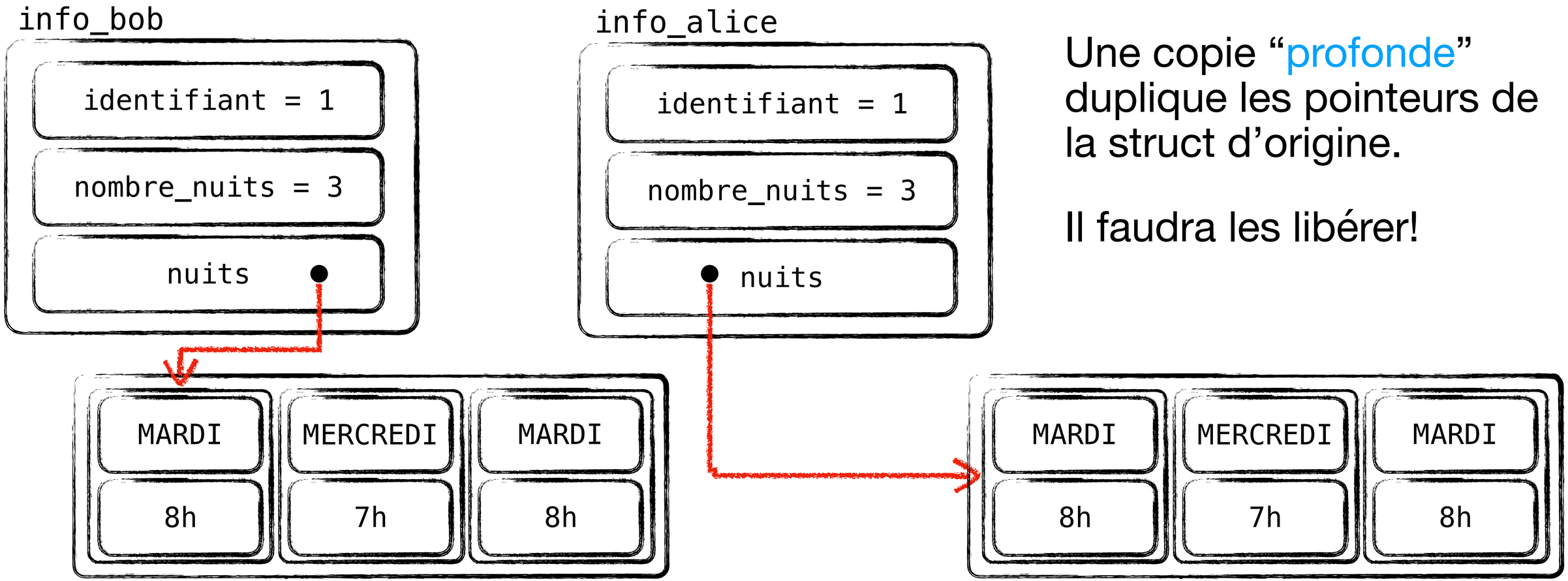
La copie superficielle pointe vers le même tableau que l'original.

Si on modifie `info_alice.nuits`, alors on modifie aussi `info_bob.nuits` !

Copie profonde

Deep copy

```
info_t info_alice = info_bob;
info_alice.nuits = malloc(3 * sizeof(sommeil_t));
info_alice.nuits[0] = info_bob.nuits[0]; etc.
```



Une copie “profonde” duplique les pointeurs de la struct d’origine.

Il faudra les libérer!