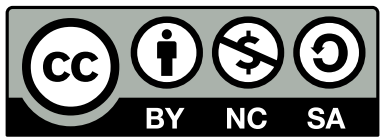
INF1135
Programmation procédurale
Chapitre 8 — Définition de types

Alexandre Blondin Massé

Université du Québec à Montréal

v263



Plan

1. Rappel et motivation
2. Énumérations
3. Structures
4. Structures et fonctions
5. Synonymes de types
6. Imbrication et tableaux
7. Champs de bits
8. Unions
9. Tests
10. Débogage

Plan

1. Rappel et motivation
2. Énumérations
3. Structures
4. Structures et fonctions
5. Synonymes de types
6. Imbrication et tableaux
7. Champs de bits
8. Unions
9. Tests
10. Débogage

Rappel : tableaux

```
double tensions[3] = {1.25, 2.50, 0.75};
```

- Un tableau regroupe des éléments de **même** type
- Accès par un **indice**, qui ne dit rien de la signification de la donnée
- Comment regrouper le nom d'un capteur, sa broche et sa tension ?
- Comment donner un nom à une valeur parmi quelques possibilités ?

Dans ce chapitre

On définit nos propres types : les **énumérations**, les **structures**, puis les **champs de bits** et les **unions**, deux outils pour contrôler la disposition en mémoire.

Motivation : nommer des valeurs

```c
int feu = 2;      // 0: rouge, 1: jaune, 2: vert
if (feu == 2) {
  printf("Avancez\n");
}
```

- Il faut se souvenir de ce que signifie 2
- Rien n'indique que `feu` n'admet que trois valeurs
- Le commentaire peut devenir faux sans que le compilateur le remarque

Motivation : regrouper des données

```
char noms[3][16]    = {"temperature", "pression", "lumiere"};
int broches[3]      = {14, 15, 16};
double tensions[3] = {1.25, 2.50, 0.75};

void afficher(const char nom[], int broche, double tension);
```

- Un même capteur est **réparti** dans trois tableaux
- Chaque information ajoutée allonge la liste des paramètres
- Pour trier, il faut échanger dans les trois tableaux, **sans en oublier**

Idée

Un seul type `Capteur`, qui contient les trois informations, et un seul tableau.

Plan

1. Rappel et motivation
2. Énumérations
3. Structures
4. Structures et fonctions
5. Synonymes de types
6. Imbrication et tableaux
7. Champs de bits
8. Unions
9. Tests
10. Débogage

Exemple : feu de circulation

Fichier `feu.c` :

```c
#include <stdio.h>

// Etat d'un feu de circulation
enum Feu { ROUGE, JAUNE, VERT };

/**
 * Affiche la consigne associee a un feu.
 *
 * @param f  L'etat du feu
 */
void afficher_consigne(enum Feu f) {
  switch (f) {
    case ROUGE: printf("Arretez\n"); break;
    case JAUNE: printf("Ralentissez\n"); break;
    case VERT:  printf("Avancez\n"); break;
  }
}

int main(void) {
  enum Feu f = JAUNE;
  printf("f vaut %d\n", f);
  afficher_consigne(f);
  f = VERT;
  printf("f vaut %d\n", f);
  afficher_consigne(f);
  return 0;
}
```

```
$ ./feu
f vaut 1
Ralentissez
f vaut 2
Avancez
```

- `enum Feu` : un nouveau type
- `ROUGE`, `JAUNE`, `VERT` : des constantes entières, qui valent 0, 1 et 2
- Le code se lit sans commentaire

Définir une énumération

```
// Mois de l'annee : la numerotation commence a 1
enum Mois { JAN = 1, FEV, MAR };      // 1, 2, 3

// Codes de retour d'un programme
enum Code { SUCCES, ERR_LECTURE = 10, ERR_ECRITURE };  // 0, 10, 11
```

- Sans valeur explicite : la précédente plus 1, en commençant à 0
- Noms en **majuscules**, comme les constantes
- Les noms sont **globaux** : pas de `ROUGE` dans deux énumérations d'une même portée

Aucune vérification

Une énumération reste un entier : `enum Feu f = 42;` est accepté.

Documenter une énumération

```
/**
 * Mode de fonctionnement d'une broche du microcontroleur.
 *
 * Apres une remise a zero, toute broche est en mode ENTREE.
 */
enum ModeBroche {
  ENTREE,      // Lecture d'un signal externe, numerique
  SORTIE,      // Niveau logique impose par le microcontroleur
  ANALOGIQUE,  // Lecture d'une tension, de 0 a 3.3 V
};
```

Une *docstring* pour le type, puis pour chaque valeur :

- Sa **signification**, et dans quels cas l'utiliser
- Ses **particularités** : valeur par défaut, unité, plage associée
- Sa valeur **numérique**, si elle est imposée de l'extérieur (registre, protocole)

Exemple : un cas oublié

Fichier `feuoubli.c` :

```c
#include <stdio.h>

enum Feu { ROUGE, JAUNE, VERT };

void afficher_consigne(enum Feu f) {
  switch (f) {
    case ROUGE: printf("Arretez\n"); break;
    case JAUNE: printf("Ralentissez\n"); break;
  }
}

int main(void) {
  afficher_consigne(VERT);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o feuoubli
      feuoubli.c
feuoubli.c:6:3: warning: enumeration
      value ‘’VERT not handled in switch
      [-Wswitch]
$ ./feuoubli
```

- Le cas `VERT` manque dans le `switch`
- Avec `-Wall`, le compilateur le signale, puisqu'il connaît toutes les valeurs du type
- Simple avertissement : le programme compile, mais n'affiche rien

Énumération ou `#define` ?

`enum`

- Valeurs regroupées et numérotées
- Cas oubliés signalés
- Noms visibles dans `gdb`
- Entiers seulement

`#define`

- Valeurs indépendantes
- Remplacement de texte, sans type
- Inconnu de `gdb`
- Tout type de valeur : `3.14`, `"abc"`

Plan

1. Rappel et motivation
2. Énumérations
3. Structures
4. Structures et fonctions
5. Synonymes de types
6. Imbrication et tableaux
7. Champs de bits
8. Unions
9. Tests
10. Débogage

Exemple : un point du plan

Fichier `point.c` :

```c
#include <stdio.h>

struct Point {
  double x;
  double y;
};

int main(void) {
  struct Point p;
  p.x = 1.5;
  p.y = -2.0;
  printf("p = (%.1f, %.1f)\n", p.x, p.y);

  p.x += 1.0;
  printf("p = (%.1f, %.1f)\n", p.x, p.y);

  printf("Taille : %zu octets\n", sizeof p);
  return 0;
}
```

```
$ ./point
p = (1.5, -2.0)
p = (2.5, -2.0)
Taille : 16 octets
```

- `struct Point` : un nouveau type, formé de deux **membres**
- `p` : une variable de ce type
- `p.x` : le membre `x` de `p`, qui s'utilise comme un `double` ordinaire

Définition et instanciation

```
struct Point {      // definition du type
  double x;
  double y;
};                  // point-virgule obligatoire

struct Point p;     // instanciation : une variable
struct Point q;     // une autre, independante
```

- La **définition** décrit le type : elle ne réserve aucune mémoire
- L'**instanciation** crée une variable, avec ses propres membres
- `Point` est l'**étiquette** ; le type se nomme `struct Point`
- Définition placée en général au début du fichier, avant les fonctions qui l'utilisent
- Un membre peut être un nombre, un tableau, un pointeur ou même une autre structure

Exemple : initialisations

Fichier `init.c` :

```c
#include <stdio.h>

struct Capteur {
  char nom[16];
  int broche;
  double tension;
};

int main(void) {
  // Dans l'ordre des membres
  struct Capteur a = {"temperature", 14, 1.25};
  // Par nom : l'ordre est libre, le reste vaut 0
  struct Capteur b = {.tension = 2.5, .nom = "pression"};
  // Tout a zero (C23)
  struct Capteur c = {};

  printf("[%s] %d %.2f\n", a.nom, a.broche, a.tension);
  printf("[%s] %d %.2f\n", b.nom, b.broche, b.tension);
  printf("[%s] %d %.2f\n", c.nom, c.broche, c.tension);
  return 0;
}
```

```
$ ./init
[temperature] 14 1.25
[pression] 0 2.50
[] 0 0.00
```

- a : valeurs dans l'ordre des membres
- b : membres désignés par leur nom
- c : tout à zéro
- Pour `nom`, zéro signifie que tous les caractères valent `'\0'` : c'est la **chaîne vide**, d'où `[]`

Initialisation

```
struct Point a = {1.0, 2.0};              // dans l'ordre
struct Point b = {.y = 2.0, .x = 1.0};    // initialiseurs designes
struct Point c = {.x = 1.0};              // c.y vaut 0.0
struct Point d = {};                      // tout a zero (C23)
struct Point e;                           // valeurs indeterminees
```

- Membres omis dans un initialiseur : mis à **zéro**
- Sans initialiseur, une variable locale contient des valeurs **indéterminées**, comme toute variable locale
- Les initialiseurs désignés restent corrects si l'ordre des membres change
- Les accolades ne servent qu'à l'**initialisation** : l'affectation `a = {3.0, 4.0};` est refusée

Exemple : copier et comparer

Fichier copy.c :

```c
struct Capteur {
  char nom[16];
  int broche;
};

int main(void) {
  struct Capteur a = {"temperature", 14};
  struct Capteur b = a;    // copie complete

  printf("Apres la copie\n");
  printf("a : %s %d\n", a.nom, a.broche);
  printf("b : %s %d\n", b.nom, b.broche);

  strcpy(b.nom, "pression");
  b.broche = 15;

  printf("Apres la modification de b\n");
  printf("a : %s %d\n", a.nom, a.broche);
  printf("b : %s %d\n", b.nom, b.broche);

  // Comparaison membre par membre
  bool est_pareil = strcmp(a.nom, b.nom) == 0
                    && a.broche == b.broche;
  printf("Identiques : %d\n", est_pareil);
  return 0;
}
```

```
$ ./copy
Apres la copie
a : temperature 14
b : temperature 14
Apres la modification de b
a : temperature 14
b : pression 15
Identiques : 0
```

- Modifier b ne change pas a
- nom n'est **pas partagé** : b reçoit sa propre copie du tableau, alors que = est refusé entre deux tableaux

Affectation et comparaison

Affectation

`b = a` copie **tous** les membres, tableaux compris.

Comparaison

== n'existe pas pour les structures :

```
structcmp.c:6:9: error: invalid operands to binary == (have ‘struct ’Point and ‘struct ’Point)
```

Il faut écrire une comparaison **membre par membre**, avec `strcmp` pour les chaînes.

Exemple : un membre pointeur

Fichier `shallow.c` :

```c
#include <stdio.h>

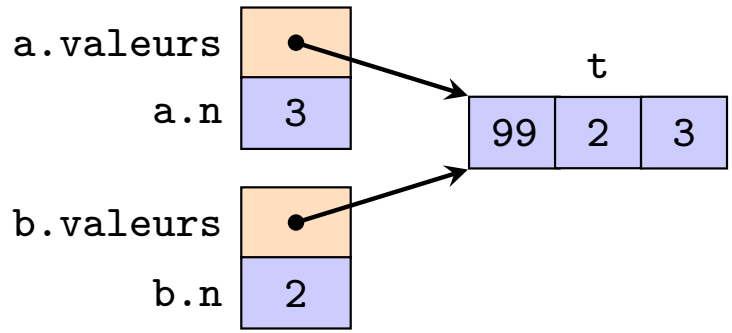
struct Serie {
  int *valeurs;   // seule l'adresse est copiee
  int n;
};

int main(void) {
  int t[3] = {1, 2, 3};
  struct Serie a = {t, 3};
  struct Serie b = a;

  b.valeurs[0] = 99;
  b.n = 2;

  printf("a.valeurs[0] = %d\n", a.valeurs[0]);
  printf("a.n = %d\n", a.n);
  printf("Meme tableau : %d\n",
         a.valeurs == b.valeurs);
  return 0;
}
```

```
$ ./shallow
a.valeurs[0] = 99
a.n = 3
Meme tableau : 1
```



- `n` : deux copies indépendantes
- `valeurs` : deux pointeurs vers le **même** tableau

Tableau ou pointeur dans une structure

Membre tableau

```
char nom[16];
```

- Stocké **dans** la structure
- Copié avec elle
- Taille fixée à la compilation

Membre pointeur

```
const char *nom;
```

- Données **hors** de la structure
- Seule l'adresse est copiée
- Les données doivent exister aussi longtemps que le pointeur

Copie superficielle

L'affectation ne suit pas les pointeurs. Pour dupliquer les données pointées, il faut le faire explicitement.

Exemple : disposition en mémoire

Fichier `layout.c` :

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

struct MalOrdonnee { char a; int n; char b; };

struct BienOrdonnee { int n; char a; char b; };

int main(void) {
  struct MalOrdonnee m;
  struct BienOrdonnee b;

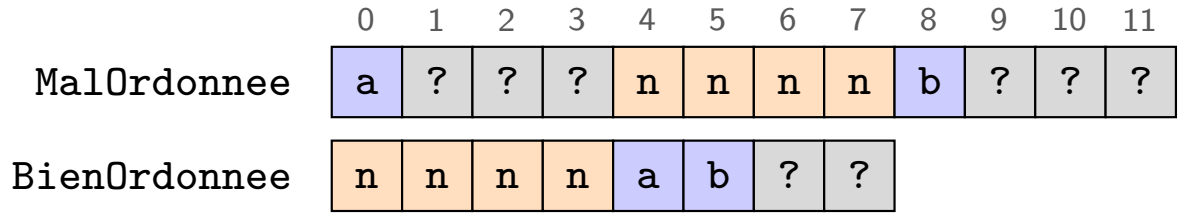
  printf("MalOrdonnee : %zu octets\n", sizeof m);
  printf("  a: %zu, n: %zu, b: %zu\n",
         offsetof(struct MalOrdonnee, a),
         offsetof(struct MalOrdonnee, n),
         offsetof(struct MalOrdonnee, b));

  printf("BienOrdonnee : %zu octets\n", sizeof b);
  printf("  n: %zu, a: %zu, b: %zu\n",
         offsetof(struct BienOrdonnee, n),
         offsetof(struct BienOrdonnee, a),
         offsetof(struct BienOrdonnee, b));
  return 0;
}
```

```
$ ./layout
MalOrdonnee : 12 octets
  a: 0, n: 4, b: 8
BienOrdonnee : 8 octets
  n: 0, a: 4, b: 5
```

- Mêmes membres, mais pas la même taille
- `offsetof` (de `<stddef.h>`) : position d'un membre, en octets
- $1 + 4 + 1 = 6$ octets de données seulement

Alignement et remplissage



- Ici, un `int` doit commencer à une adresse **multiple de 4** : c'est l'**alignement**, qui rend l'accès possible ou plus rapide selon le processeur
- Le compilateur insère des octets de **remplissage** (en gris), jamais avant le premier membre
- Du remplissage est aussi ajouté à la fin, pour que les éléments d'un **tableau** restent alignés
- Les membres restent dans l'ordre de leur déclaration

À retenir

`sizeof` d'une structure peut dépasser la somme de ses membres : toujours utiliser `sizeof`, ne jamais additionner. Placer les grands membres d'abord réduit le remplissage.

Documenter une structure

```
/**
 * Un capteur analogique relie au microcontroleur.
 *
 * Une tension negative indique qu'aucune mesure n'a ete prise.
 */
struct Capteur {
  char nom[16];     // Nom, chaine terminee par '\0' (15 car. max)
  int broche;       // Numero de broche, unique, de 0 a 31
  double tension;   // Derniere mesure en volts, de 0 a 3.3
};
```

Une *docstring* pour le type, puis pour chaque membre :

- Son **rôle**
- Son **unité** : des volts ou des millivolts ? des secondes ou des millisecondes ?
- Ses **contraintes** : plage de valeurs, longueur maximale, valeurs particulières

Plan

1. Rappel et motivation
2. Énumérations
3. Structures
4. Structures et fonctions
5. Synonymes de types
6. Imbrication et tableaux
7. Champs de bits
8. Unions
9. Tests
10. Débogage

Exemple : passage par valeur

Fichier `byvalue.c` :

```c
struct Point {
  double x;
  double y;
};

// Ne modifie que la copie recue
void deplacer(struct Point p, double dx, double dy) {
  p.x += dx;
  p.y += dy;
}

// Retourne une structure par valeur
struct Point milieu(struct Point a, struct Point b) {
  struct Point m = {(a.x + b.x) / 2, (a.y + b.y) / 2};
  return m;
}

int main(void) {
  struct Point p = {1.0, 2.0};
  struct Point q = {3.0, 6.0};

  deplacer(p, 10.0, 10.0);
  printf("p = (%.1f, %.1f)\n", p.x, p.y);

  struct Point m = milieu(p, q);
  printf("m = (%.1f, %.1f)\n", m.x, m.y);
  return 0;
}
```

```
$ ./byvalue
p = (1.0, 2.0)
m = (2.0, 4.0)
```

- `deplacer` modifie une **copie** : `p` n'a pas changé
- `milieu` **retourne** une structure, elle aussi copiée

Exemple : passage par adresse

Fichier `translate.c` :

```
struct Point {
  double x;
  double y;
};

// Avec le dereferencement explicite
void deplacer1(struct Point *p, double dx, double dy) {
  (*p).x += dx;
  (*p).y += dy;
}

// La meme fonction avec ->
void deplacer2(struct Point *p, double dx, double dy) {
  p->x += dx;
  p->y += dy;
}

int main(void) {
  struct Point p = {1.0, 2.0};

  deplacer1(&p, 10.0, 10.0);
  printf("p = (%.1f, %.1f)\n", p.x, p.y);

  deplacer2(&p, 0.5, 0.5);
  printf("p = (%.1f, %.1f)\n", p.x, p.y);
  return 0;
}
```

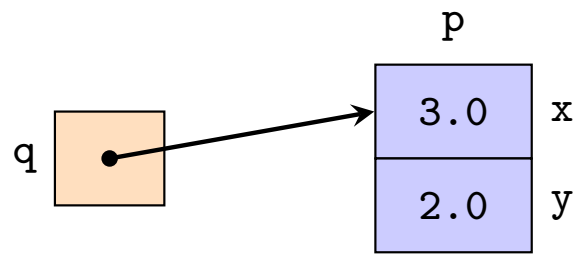
```
$ ./translate
p = (11.0, 12.0)
p = (11.5, 12.5)
```

- Les deux fonctions font **exactement** la même chose
- `p->x` est plus lisible que `(*p).x`

L'opérateur ->

```
struct Point p = {1.0, 2.0};
struct Point *q = &p;

p.x     = 3.0;  // structure
(*q).x  = 3.0;  // equivalent
q->x    = 3.0;  // equivalent
```



- `.` s'applique à une **structure** ; `->` s'applique à un **pointeur** vers une structure
- Les parenthèses de `(*q).x` sont nécessaires : `.` est prioritaire sur `*`
- `*q.x` signifie `*(q.x)`, refusé puisque `q` n'est pas une structure

Valeur, adresse ou adresse constante ?

```
double norme2(struct Point p);                    // petite structure
void afficher(const struct Capteur *c);           // lecture seulement
void deplacer(struct Point *p, double dx, double dy);  // modification
```

- **Valeur** : simple, mais copie toute la structure à chaque appel
- **Adresse** : paramètre de **sortie**, pour modifier la structure de l'appelant
- **Adresse constante** : pas de copie, et le compilateur refuse toute modification

Retour

Retourner une structure par valeur est correct. Retourner l'**adresse** d'une structure locale ne l'est pas : comme vu précédemment, elle cesse d'exister à la fin de la fonction.

Modifier une structure : paramètre de sortie

```
struct Point deplacee(struct Point p, double dx, double dy);
void deplacer(struct Point *p, double dx, double dy);

p = deplacee(p, 1.0, 2.0);  // copie a l'appel et au retour
deplacer(&p, 1.0, 2.0);     // parametre de sortie : aucune copie
```

Pour modifier une structure existante, il est préférable d'utiliser un paramètre de sortie :

- **Aucune copie**, quelle que soit la taille de la structure
- L'intention est **visible à l'appel** : `&p` indique que `p` peut changer
- Pas d'oubli possible : `deplacee(p, 1.0, 2.0);` sans affectation perd le résultat, sans avertissement

Et le retour par valeur ?

Il convient pour **créer** une nouvelle structure, comme `milieu`, en particulier si elle est petite.

Plan

1. Rappel et motivation
2. Énumérations
3. Structures
4. Structures et fonctions
5. Synonymes de types
6. Imbrication et tableaux
7. Champs de bits
8. Unions
9. Tests
10. Débogage

Exemple : alléger les déclarations

Fichier `typedef.c` :

```c
#include <stdio.h>

typedef struct {
  double x;
  double y;
} Point;

Point milieu(Point a, Point b) {
  Point m = {(a.x + b.x) / 2, (a.y + b.y) / 2};
  return m;
}

void afficher(const Point *p) {
  printf("(%.1f, %.1f)\n", p->x, p->y);
}

int main(void) {
  Point p = {1.0, 2.0};
  Point q = {3.0, 6.0};
  Point m = milieu(p, q);
  afficher(&m);
  return 0;
}
```

```
$ ./typedef
(2.0, 4.0)
```

- `Point` remplace `struct Point` partout
- Déclarations et prototypes plus courts
- Le comportement est identique

Définir un synonyme avec typedef

```
typedef double Volt;                        // un type de base
typedef struct { double x, y; } Point;      // structure sans etiquette
typedef struct Capteur Capteur;             // meme nom permis
typedef enum { ROUGE, JAUNE, VERT } Feu;
```

- `typedef type synonyme;` donne un **autre nom** à un type existant
- Pas de nouveau type : `Volt` et `double` se mélangent sans avertissement, le synonyme ne sert qu'à documenter
- Déjà rencontrés : `size_t`, `uint8_t`, etc.
- Dans ce cours : **majuscule initiale** pour tout type défini par le programmeur : étiquette de structure, d'union ou d'énumération, et synonyme
- L'étiquette reste nécessaire si la structure contient un pointeur vers son propre type

Plan

1. Rappel et motivation
2. Énumérations
3. Structures
4. Structures et fonctions
5. Synonymes de types
6. Imbrication et tableaux
7. Champs de bits
8. Unions
9. Tests
10. Débogage

Exemple : structures imbriquées

Fichier `nested.c` :

```c
#include <stdio.h>

typedef struct {
  double x;
  double y;
} Point;

typedef struct {
  Point debut;
  Point fin;
} Segment;

Point milieu(const Segment *s) {
  Point m = {(s->debut.x + s->fin.x) / 2,
             (s->debut.y + s->fin.y) / 2};
  return m;
}

int main(void) {
  Segment s = {{0.0, 0.0}, {4.0, 2.0}};
  s.fin.x = 6.0;

  Point m = milieu(&s);
  printf("milieu = (%.1f, %.1f)\n", m.x, m.y);
  printf("%zu = 2 x %zu octets\n",
         sizeof s, sizeof s.debut);
  return 0;
}
```

```
$ ./nested
milieu = (3.0, 1.0)
32 = 2 x 16 octets
```

- Un `Segment` contient deux `Point`, stockés dans la structure
- Accès en chaîne, de gauche à droite : `s.fin.x`
- Seul le premier niveau utilise `->` avec un pointeur : `s->debut.x`
- Accolades imbriquées pour l'initialisation

Exemple : tableau de structures

Fichier `sensors.c` :

```c
#include <stdio.h>

typedef struct {
  char nom[16];
  int broche;
  double tension;
} Capteur;

void afficher(const Capteur t[], int n) {
  for (int i = 0; i < n; i++) {
    printf("%-12s %2d %5.2f V\n",
           t[i].nom, t[i].broche,
           t[i].tension);
  }
}

double moyenne(const Capteur t[], int n) {
  double somme = 0.0;
  for (int i = 0; i < n; i++) {
    somme += t[i].tension;
  }
  return somme / n;
}
```

```c
int main(void) {
  Capteur capteurs[] = {
    {"temperature", 14, 1.25},
    {"pression", 15, 2.50},
    {"lumiere", 16, 0.75},
  };
  int n = sizeof capteurs / sizeof capteurs[0];

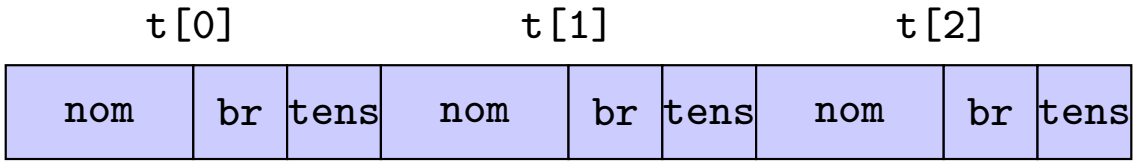
  afficher(capteurs, n);
  printf("Moyenne : %.2f V\n",
         moyenne(capteurs, n));
  return 0;
}
```

```
$ ./sensors
temperature  14  1.25 V
pression     15  2.50 V
lumiere      16  0.75 V
Moyenne : 1.50 V
```

- Un seul tableau : chaque capteur reste **groupé**
- Ajouter un membre ne change pas les prototypes

Tableaux de structures

```
Capteur t[3];
t[1].tension = 2.5;              // membre de l'element 1
t[1].nom[0] = 'P';               // 1er caractere de son nom
(t + 1)->tension = 2.5;          // equivalent a t[1].tension
```



- Éléments **contigus** ; `t + 1` avance de `sizeof(Capteur)` octets
- Nombre d'éléments : `sizeof t / sizeof t[0]`, dans la fonction où `t` est déclaré
- En paramètre : l'adresse du premier élément, et la taille à part, comme tout tableau

Exemple : recherche par nom

Fichier `find.c` :

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

typedef struct {
  char nom[16];
  double tension;
} Capteur;

/**
 * Cherche un capteur par son nom.
 *
 * @param t    Le tableau de capteurs
 * @param n    Le nombre de capteurs
 * @param nom  Le nom cherche
 * @return     L'adresse trouvee, NULL si absent
 */
Capteur *chercher(Capteur t[], int n,
                  const char *nom) {
  for (int i = 0; i < n; i++) {
    if (strcmp(t[i].nom, nom) == 0) return &t[i];
  }
  return NULL;
}
```

- Retourne l'**adresse** de l'élément : pas de copie, et l'appelant peut le modifier
- Correct : le tableau existe encore après l'appel
- `NULL` si absent

Exemple : recherche par nom (suite)

```c
int main(void) {
  Capteur capteurs[] = {
    {"temperature", 1.25},
    {"pression", 2.50},
    {"lumiere", 0.75},
  };

  Capteur *c = chercher(capteurs, 3, "pression");
  if (c != NULL) {
    c->tension += 0.10;   // modifie le tableau
  }
  printf("%s : %.2f V\n",
         capteurs[1].nom, capteurs[1].tension);

  c = chercher(capteurs, 3, "vibration");
  printf("vibration trouve : %d\n", c != NULL);
  return 0;
}
```

```
$ ./find
pression : 2.60 V
vibration trouve : 0
```

- Vérifier `NULL` **avant** d'utiliser `->`
- `c->tension` modifie bien l'élément du tableau

Exemple : trier des structures

Fichier `sort.c` :

```c
#include <stdio.h>

typedef struct {
  char nom[16];
  double tension;
} Capteur;

void echanger(Capteur *a, Capteur *b) {
  Capteur temp = *a;
  *a = *b;
  *b = temp;
}

// Tri par selection, plus forte tension d'abord
void trier(Capteur t[], int n) {
  for (int i = 0; i < n - 1; i++) {
    int m = i;
    for (int j = i + 1; j < n; j++) {
      if (t[j].tension > t[m].tension) m = j;
    }
    echanger(&t[i], &t[m]);
  }
}
```

- Même tri par sélection que pour un tableau d'entiers
- La **clé** de comparaison est un membre
- `echanger` déplace toute la structure en trois affectations : le nom suit la tension

Exemple : trier des structures (suite)

```
int main(void) {
  Capteur capteurs[] = {
    {"temperature", 1.25},
    {"pression", 2.50},
    {"lumiere", 0.75},
  };

  trier(capteurs, 3);
  for (int i = 0; i < 3; i++) {
    printf("%-12s %5.2f V\n",
           capteurs[i].nom, capteurs[i].tension);
  }
  return 0;
}
```

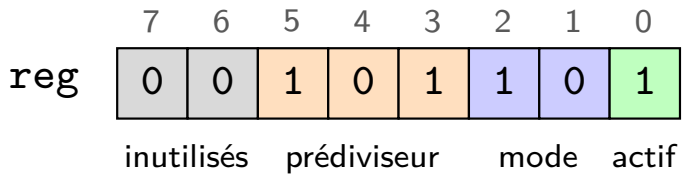
```
$ ./sort
pression      2.50 V
temperature   1.25 V
lumiere       0.75 V
```

Plan

1. Rappel et motivation
2. Énumérations
3. Structures
4. Structures et fonctions
5. Synonymes de types
6. Imbrication et tableaux
7. Champs de bits
8. Unions
9. Tests
10. Débogage

Exemple : le registre d'un minuteur

Un **minuteur** de microcontrôleur compte les tics d'une horloge, pour mesurer une durée ou déclencher une action périodique. Son **registre de contrôle** le configure :



- **actif** (bit 0) : 1 si le minuteur est en marche, 0 s'il est arrêté
- **mode** (bits 1 et 2) : la façon de compter

  0 : une seule fois, puis arrêt 1 : en boucle, en repartant de zéro
  2 : à rebours, jusqu'à zéro 3 : en montant, puis en descendant
- **prédiviseur** (bits 3 à 5) : la valeur p ne compte qu'un tic sur 2^p

À quoi sert le prédiviseur ?

Le compteur atteint vite sa valeur maximale, car l'horloge est très rapide. Compter moins de tics permet de mesurer des durées **plus longues**, mais **moins finement**. Avec une horloge de 1 MHz et un compteur de 16 bits :
$p = 0$: précision de 1 µs, jusqu'à environ 65 ms
$p = 5$: précision de 32 µs, jusqu'à environ 2 s

Exemple : interpréter le registre du minuteur

Chaque valeur du registre décrit une configuration complète (les bits 6 et 7 restent à 0) :

Valeur	Prédiviseur	Mode	Actif	Interprétation
0x00	000	00	0	Arrêté
0x01	000	00	1	En marche, une seule fois, chaque tic
0x03	000	01	1	En marche, en boucle, chaque tic
0x1F	011	11	1	En marche, montée et descente, 1 tic sur 8
0x2D	101	10	1	En marche, à rebours, 1 tic sur 32
0x3A	111	01	0	Arrêté, prêt : en boucle, 1 tic sur 128

- 0x2D est la valeur du schéma précédent
- Quand **actif** vaut 0, le minuteur ne compte pas, mais conserve sa configuration

Un modèle simplifié

Les registres d'un vrai microcontrôleur sont décrits dans sa **fiche technique**.

Rappel : registres et masques

Avec les masques vus précédemment, pour démarrer le minuteur et choisir son mode :

```
#define ACTIVE_MASK (1 << 0)          // bit 0
#define MODE_POS    1                 // bits 1 et 2
#define MODE_MASK   (3 << MODE_POS)   // 3 : deux bits a 1

unsigned int reg = 0x02;  // arrete, en mode 1
reg |= ACTIVE_MASK;       // demarrer
reg &= ~MODE_MASK;        // remettre les bits du mode a 0
reg |= 2 << MODE_POS;     // ecrire le mode 2
unsigned int mode = (reg & MODE_MASK) >> MODE_POS;   // lire le mode
```

- Efficace et précis, mais difficile à relire
- Une erreur de masque ou de décalage modifie les bits **voisins**
- Peut-on nommer ces bits, comme des membres ?

Exemple : registre de contrôle

Fichier `bitfield.c` :

```c
#include <stdio.h>

// Registre de controle d'un minuteur
typedef struct {
  unsigned int actif : 1;         // 0 ou 1
  unsigned int mode : 2;          // de 0 a 3
  unsigned int prediviseur : 3;   // de 0 a 7
} Controle;

int main(void) {
  Controle c = {};

  c.actif = 1;
  c.mode = 2;
  c.prediviseur = 5;
  printf("actif=%u mode=%u prediviseur=%u\n",
         c.actif, c.mode, c.prediviseur);

  c.mode = 3;
  printf("mode=%u prediviseur=%u\n",
         c.mode, c.prediviseur);

  printf("Taille : %zu octets\n", sizeof c);
  return 0;
}
```

```
$ ./bitfield
actif=1 mode=2 prediviseur=5
mode=3 prediviseur=5
Taille : 4 octets
```

- `:` 2 : ce membre occupe **2 bits**
- Lecture et écriture comme un membre ordinaire
- Les décalages et les masques sont générés par le compilateur
- Modifier `mode` ne touche pas `prediviseur`

Déclaration et règles

```
struct Controle {
  unsigned int mode : 2;    // Mode de comptage, de 0 a 3
  bool est_actif : 1;       // Minuteur en marche
};
```

- Types garantis : `unsigned int`, `int`, `bool`
- n bits non signés : valeurs de 0 à $2^n - 1$; préciser leur signification dans un commentaire
- Valeur trop grande : seuls les bits de poids faible sont conservés
- Pas d'adresse : un bit n'a pas d'adresse propre, donc ni `&`, ni `scanf`

```
bitlimits.c:7:12: warning: unsigned conversion from ‘’int to ‘unsigned char’:2 changes value
      from ‘’5 to ‘’1 [-Woverflow]
bitlimits.c:8:21: error: cannot take address of bit-field ‘’mode
```

Champs de bits ou masques ?

	Champs de bits	**Masques**
Lisibilité	Membres nommés	Opérations explicites
Accès	Généré par le compilateur	Écrit à la main
Ordre des bits	Dépend du compilateur	Fixé par le code
Portabilité	Limitée	Totale
Plusieurs champs à la fois	Non	Oui

Quel outil choisir ?

- **Champs de bits** : économiser de la mémoire dans le programme, quand la disposition exacte importe peu
- **Masques** : registres matériels, formats de fichiers, protocoles réseau, où chaque bit a une position **imposée**

Disposition

L'ordre des bits et le remplissage dépendent de l'implémentation : à vérifier pour chaque compilateur visé.

Plan

1. Rappel et motivation
2. Énumérations
3. Structures
4. Structures et fonctions
5. Synonymes de types
6. Imbrication et tableaux
7. Champs de bits
8. Unions
9. Tests
10. Débogage

Exemple : deux lectures des mêmes octets

Fichier `union.c` :

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

union Valeur {
  float reel;
  uint32_t entier;
};

int main(void) {
  union Valeur v;

  v.reel = 1.0f;
  printf("reel   : %.1f\n", v.reel);
  printf("entier : 0x%08X\n", v.entier);

  v.entier = 0x40490FDB;
  printf("reel   : %f\n", v.reel);

  printf("Adresses : %p %p\n",
         (void *) &v.reel, (void *) &v.entier);
  printf("Taille : %zu octets\n", sizeof v);
  return 0;
}
```

```
$ ./union
reel   : 1.0
entier : 0x3F800000
reel   : 3.141593
Adresses : 0x7ffea7d05454 0
      x7ffea7d05454
Taille : 4 octets
```

- Les deux membres ont la **même adresse**
- Écrire `reel` modifie `entier`, et inversement
- 0x3F800000 : la représentation binaire de `1.0f`

Définition et usage

```
union Valeur {
  float reel;
  uint32_t entier;
};

union Valeur a = {1.0f};                // premier membre
union Valeur b = {.entier = 0x10};      // membre designe
```

- Même syntaxe qu'une structure : `.`, `->`, `typedef`, affectation
- Les membres **partagent** le même emplacement : écrire l'un écrase les autres
- Lire un autre membre que le dernier écrit **réinterprète** ses octets ; le résultat dépend de la machine
- Usages : économiser de la mémoire quand une seule donnée est valide à la fois, et examiner la représentation d'une valeur

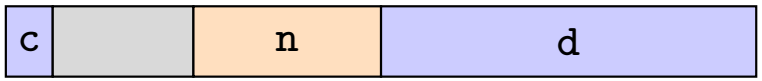
struct ou union : la mémoire

```
struct S {
  char c;
  int n;
  double d;
};

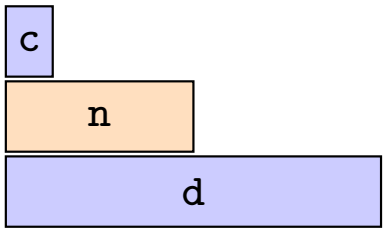
union U {
  char c;
  int n;
  double d;
};
```

```
$ ./sizes
struct : 16 octets
union  : 8 octets
```

struct S : membres **successifs**



union U : membres **superposés**



struct

Taille : tous les membres, plus le remplissage. **Tous** les membres sont valides en même temps.

union

Taille : le plus grand membre, plus le remplissage. **Un seul** membre est valide à la fois.

Exemple : voir les octets d'une valeur

Fichier `bytes.c` :

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

union Mot {
  uint32_t valeur;
  uint8_t octets[4];
};

int main(void) {
  union Mot m = {.valeur = 0x11223344};

  for (int i = 0; i < 4; i++) {
    printf("octets[%d] = 0x%02X\n", i, m.octets[i]);
  }

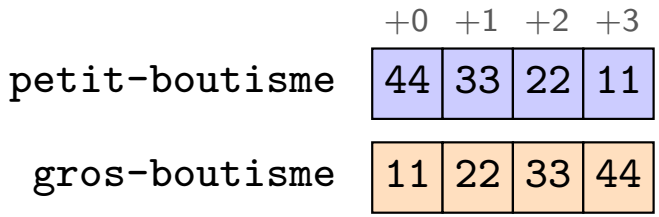
  m.octets[0] = 0xFF;
  printf("valeur = 0x%08X\n", m.valeur);
  return 0;
}
```

```
$ ./bytes
octets[0] = 0x44
octets[1] = 0x33
octets[2] = 0x22
octets[3] = 0x11
valeur = 0x112233FF
```

- Sur cette machine, l'octet de **poids faible** (0x44) vient en premier : c'est le **petit-boutisme** (*little-endian*)
- Le **gros-boutisme** (*big-endian*) donnerait l'ordre inverse
- Un programme qui échange des données binaires doit tenir compte du **boutisme** (*endianness*)

Le boutisme

Le **boutisme** est l'ordre des octets d'une valeur en mémoire. Pour `0x11223344` :



Plateforme	**Processeur**	**Boutisme**
Windows, macOS, Linux (ordinateurs)	x86-64, ARM	Petit
Android, iOS	ARM	Petit
Microcontrôleurs : Arduino Uno, STM32, ESP32	AVR, ARM, Xtensa, RISC-V	Petit
Unix de serveurs : AIX, Solaris	POWER, SPARC	Gros
Linux sur ordinateur central	IBM Z	Gros
Protocoles réseau (TCP/IP)	Sans objet	Gros

Une question de processeur

Le boutisme dépend du **processeur**, pas du système d'exploitation : Linux existe dans les deux cas. Le petit-boutisme domine aujourd'hui, mais un fichier ou un message échangé doit fixer son ordre, indépendamment de la machine. Sous Linux, `lscpu` l'indique (`Byte Order`).

Exemple : un registre, deux vues

Fichier `register.c` :

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

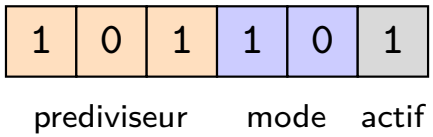
typedef union {
  uint32_t brut;
  struct {
    unsigned int actif : 1;
    unsigned int mode : 2;
    unsigned int prediviseur : 3;
  } champs;
} Registre;

int main(void) {
  Registre r = {.brut = 0};

  r.champs.actif = 1;
  r.champs.mode = 2;
  r.champs.prediviseur = 5;
  printf("brut = 0x%08X\n", r.brut);

  r.brut = 0x03;
  printf("actif=%u mode=%u prediviseur=%u\n",
         r.champs.actif, r.champs.mode,
         r.champs.prediviseur);
  return 0;
}
```

```
$ ./register
brut = 0x0000002D
actif=1 mode=1 prediviseur=0
```



- `brut` : tout le registre d'un coup
- `champs` : bit par bit, par nom
- 0x2D = 101101 en binaire
- Ce résultat dépend du compilateur : ici, le premier membre occupe le bit de poids faible

Exemple : union étiquetée

Fichier shape.c :

```c
#include <stdio.h>

#define PI 3.14159

enum TypeForme { CERCLE, RECTANGLE };

/**
 * Une forme geometrique, mesuree en cm.
 */
typedef struct {
  enum TypeForme type;    // Membre valide
  union {
    double rayon;         // Si CERCLE
    struct {
      double largeur;     // Si RECTANGLE
      double hauteur;     // Si RECTANGLE
    } dims;
  };
} Forme;
```

```c
double aire(const Forme *f) {
  switch (f->type) {
    case CERCLE:
      return PI * f->rayon * f->rayon;
    case RECTANGLE:
      return f->dims.largeur * f->dims.hauteur;
  }
  return 0.0;
}

int main(void) {
  Forme formes[] = {
    {CERCLE, {.rayon = 1.0}},
    {RECTANGLE, {.dims = {2.0, 3.0}}},
  };

  for (int i = 0; i < 2; i++) {
    printf("Aire : %.2f\n", aire(&formes[i]));
  }
  printf("Taille : %zu octets\n",
         sizeof(Forme));
  return 0;
}
```

```
$ ./shape
Aire : 3.14
Aire : 6.00
Taille : 24 octets
```

Union étiquetée

- Une union ne retient **pas** quel membre est valide : c'est au programme de le faire
- Patron courant : une structure contenant une **énumération** et une **union**
- Ici, l'**étiquette** est le membre qui indique la variante active (`type`), et non l'étiquette d'un type (`Point` dans `struct Point`)
- L'étiquette et le membre écrit doivent rester **synchronisés**, sinon on lit des octets sans signification
- Si plusieurs variantes partagent des données (nom, couleur, etc.), elles vont dans la structure, hors de l'union

Documenter une union

Comme une structure (rôle, unité, contraintes), en précisant pour chaque membre **dans quel cas** il est valide : `// Si CERCLE`.

Quand préférer une `struct` ?

Dès que plusieurs membres sont valides en même temps, ou si le gain de mémoire est négligeable. Une union n'est pas un réflexe : elle ajoute une source d'erreurs.

Sans étiquette ou anonyme ?

```
typedef struct { double x, y; } Point;  // structure sans etiquette
typedef enum { ROUGE, VERT } Feu;       // enumeration sans etiquette

typedef struct {
  enum TypeForme type;
  union {             // union anonyme : f->rayon
    double rayon;
    struct {          // sans etiquette, membre nomme : f->dims.largeur
      double largeur;
      double hauteur;
    } dims;
  };
} Forme;
```

- **Sans étiquette** : rien après `struct`, `union` ou `enum` ; le type s'utilise alors par un `typedef` ou un membre nommé
- **Membre anonyme** : structure ou union interne, sans étiquette ni nom de membre ; ses membres s'utilisent directement
- Pas d'énumération anonyme : ses constantes s'utilisent déjà directement

Plan

1. Rappel et motivation
2. Énumérations
3. Structures
4. Structures et fonctions
5. Synonymes de types
6. Imbrication et tableaux
7. Champs de bits
8. Unions
9. Tests
10. Débogage

Exemple : tester la validation d'une date

Chaque cas de test devient un élément de tableau :

```c
#include <stdio.h>

typedef struct {
  int annee;
  int mois;    // de 1 a 12
  int jour;    // de 1 a 31, selon le mois
} Date;

// Un cas de test : date et resultat attendu
typedef struct {
  const char *nom;
  Date date;
  bool attendu;
} CasTest;

bool est_bissextile(int annee) {
  return annee % 4 == 0;
}

bool est_valide(Date d) {
  int jours[] = {31, 28, 31, 30, 31, 30,
                 31, 31, 30, 31, 30, 31};
  if (d.mois < 1 || d.mois > 12) return false;
  int max = jours[d.mois - 1];
  if (d.mois == 2 && est_bissextile(d.annee))
    max = 29;
  return d.jour >= 1 && d.jour <= max;
}
```

```c
int main(void) {
  CasTest cas[] = {
    {"ordinaire",  {2026, 11, 3}, true},
    {"mois 0",     {2026, 0, 10}, false},
    {"mois 13",    {2026, 13, 1}, false},
    {"jour 0",     {2026, 5, 0}, false},
    {"31 avril",   {2026, 4, 31}, false},
    {"31 dec.",    {2026, 12, 31}, true},
    {"29 fev. 2024", {2024, 2, 29}, true},
    {"29 fev. 2026", {2026, 2, 29}, false},
    {"29 fev. 2000", {2000, 2, 29}, true},
    {"29 fev. 1900", {1900, 2, 29}, false},
  };
  int n = sizeof cas / sizeof cas[0];

  int reussis = 0;
  for (int i = 0; i < n; i++) {
    bool obtenu = est_valide(cas[i].date);
    bool ok = obtenu == cas[i].attendu;
    printf("%-5s %-12s %d (attendu %d)\n",
           ok ? "OK" : "ECHEC", cas[i].nom,
           obtenu, cas[i].attendu);
    reussis += ok;
  }
  printf("%d/%d reussis\n", reussis, n);
  return 0;
}
```

Exécution des tests

```
$ ./testdate
OK    ordinaire    1 (attendu 1)
OK    mois 0       0 (attendu 0)
OK    mois 13      0 (attendu 0)
OK    jour 0       0 (attendu 0)
OK    31 avril     0 (attendu 0)
OK    31 dec.      1 (attendu 1)
OK    29 fev. 2024 1 (attendu 1)
OK    29 fev. 2026 0 (attendu 0)
OK    29 fev. 2000 1 (attendu 1)
ECHEC 29 fev. 1900 1 (attendu 0)
9/10 reussis
```

- Échec : une année divisible par 100 n'est bissextile que si elle est aussi divisible par 400

```
return annee % 4 == 0;                                          // incorrect
return annee % 4 == 0 && (annee % 100 != 0 || annee % 400 == 0); // correct
```

Tests dans un tableau

Ajouter un cas ne demande qu'une ligne, et la boucle de vérification reste unique.

Cas limites

Structures

- Vérifier **chaque contrainte** documentée, à ses deux bornes (`mois` à 0, 1, 12, 13)
- Combinaisons de membres : 31 avril, 29 février selon l'année
- Membres chaîne : vide, ou de longueur maximale
- Comparer les résultats **membre par membre**, puisque == n'existe pas

Tableaux de structures

- Fonction de recherche : premier élément, dernier, absent (retour `NULL`), tableau vide

Champs de bits et unions

- Valeur maximale de chaque champ (3 pour 2 bits) et effet sur les champs **voisins**
- **Chaque** valeur de l'étiquette d'une union, et une valeur invalide

Plan

1. Rappel et motivation
2. Énumérations
3. Structures
4. Structures et fonctions
5. Synonymes de types
6. Imbrication et tableaux
7. Champs de bits
8. Unions
9. Tests
10. Débogage

Inspecter une structure avec gdb

```
$ gdb -q sensors
(gdb) break 33
Breakpoint 1 at 0x12fe: file sensors.c, line 33.
(gdb) run
Breakpoint 1, main () at sensors.c:33
33    afficher(capteurs, n);
(gdb) print capteurs[1]
$1 = {nom = "pression\000\000\000\000\000\000\000", broche = 15, tension = 2.5}
(gdb) print capteurs[1].nom
$2 = "pression\000\000\000\000\000\000\000"
(gdb) print capteurs[1].tension
$3 = 2.5
(gdb) set print pretty on
(gdb) print capteurs[0]
$4 = {
  nom = "temperature\000\000\000\000",
  broche = 14,
  tension = 1.25
}
```

- `print` : tous les membres, ou un seul avec la même syntaxe qu'en C
- `set print pretty on` : un membre par ligne
- Les \000 sont les octets restants de `nom`

Structure reçue par pointeur

```
$ gdb -q sensors
(gdb) break moyenne
Breakpoint 1 at 0x1203: file sensors.c, line 18.
(gdb) run
Breakpoint 1, moyenne (t=0x7fffffffd570, n=3) at sensors.c:18
18    double somme = 0.0;
(gdb) print t
$1 = (const Capteur *) 0x7fffffffd570
(gdb) print t->nom
$2 = "temperature\000\000\000\000"
(gdb) print t[2].tension
$3 = 0.75
(gdb) print *t@2
$4 = {{nom = "temperature\000\000\000\000", broche = 14, tension = 1.25}, {nom = "pression
      \000\000\000\000\000\000\000", broche = 15, tension = 2.5}}
```

- `print t` : seulement une **adresse**
- `t->nom`, `t[2].tension` : même syntaxe qu'en C
- `print *t@2` : les 2 premiers éléments

Voir le remplissage avec gdb

```
$ gdb -q layout
(gdb) ptype /o struct MalOrdonnee
/* offset      |    size */  type = struct MalOrdonnee {
/*      0      |       1 */    char a;
/* XXX  3-byte hole      */
/*      4      |       4 */    int n;
/*      8      |       1 */    char b;
/* XXX  3-byte padding   */
                               /* total size (bytes):   12 */
                             }
(gdb) ptype /o struct BienOrdonnee
/* offset      |    size */  type = struct BienOrdonnee {
/*      0      |       4 */    int n;
/*      4      |       1 */    char a;
/*      5      |       1 */    char b;
/* XXX  2-byte padding   */
                               /* total size (bytes):    8 */
                             }
```

- `ptype /o` : position et taille de chaque membre, avec les trous de remplissage

Inspecter une union avec gdb

```
$ gdb -q union
(gdb) break 16
Breakpoint 1 at 0x11d5: file union.c, line 16.
(gdb) run
Breakpoint 1, main () at union.c:16
16    v.entier = 0x40490FDB;
(gdb) print v
$1 = {reel = 1, entier = 1065353216}
(gdb) print/x v.entier
$2 = 0x3f800000
(gdb) ptype v
type = union Valeur {
    float reel;
    uint32_t entier;
}
(gdb) x/4xb &v
0x7fffffffd5d4: 0x00  0x00  0x80  0x3f
```

- `print v` : **tous** les membres, même ceux qui ne sont pas valides ; gdb ne sait pas lequel l'est
- `print/x` : affichage en hexadécimal
- `x/4xb &v` : les 4 octets bruts, ici en petit-boutisme

Exemple : débordement dans une structure

Fichier `overflow.c` :

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

typedef struct {
  char nom[8];
  int broche;
} Capteur;

int main(void) {
  Capteur c = {"temp", 14};
  const char *saisie = "temperature";

  strcpy(c.nom, saisie);
  printf("%s : %d\n", c.nom, c.broche);
  return 0;
}
```

```
$ ./overflow
temperature : 6648437
```

- `"temperat"` remplit `nom` ; `"ure\0"` déborde sur `broche`
- $6648437 = 0\text{x}00657275$: les codes de `'\0'`, `e`, `r`, `u`, en petit-boutisme

Détecter ce débordement

Le compilateur l'a vu ici, puisque la chaîne est connue d'avance :

```
overflow.c:13:3: warning: ‘’strcpy writing 12 bytes into a region of size 8 overflows the
      destination
```

Avec une saisie au clavier, il ne pourrait pas. Le désinfecteur d'adresses ne signale rien non plus :

```
$ gcc -Wall -Wextra -std=c23 -fsanitize=address -g -o overflow overflow.c
$ ./overflow
temperature : 6648437
```

- `-fsanitize=address` surveille les limites de la **variable** `c`, pas celles de ses membres : on reste dans `c`
- Correction : `snprintf(c.nom, sizeof c.nom, "%s", saisie)`, qui tronque au besoin
- Symptôme typique : un membre change de valeur sans qu'on l'ait modifié