Programmation impérative

ENSIIE

Semestre 1 — 2020–21

Types complexes

Type énuméré

Ensemble (fini) de constantes distinctes

enum *nom_enum* { *nom_cte*$_1$, ... , *nom_cte*$_n$ };

En pratique : comme des `int`, conversions implicites depuis et vers les entiers

Possibilité de spécifier certaines valeurs

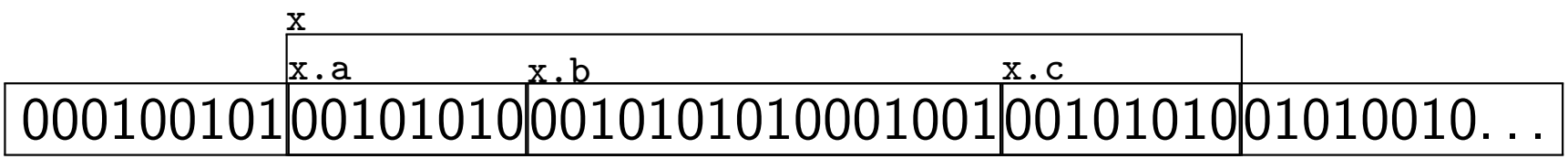
```
enum alignement
  { good, neutral, evil = 666 };
```

Le nom du nouveau type est `enum` *nom_enum*
$\Rightarrow$ Déclaration d'une variable de type énuméré :

enum *nom_enum nom_var*;

Enregistrement

- Regrouper plusieurs données dans une seule variable
- $\sim n$-uplets avec noms



- `x.a`, `x.b`, `x.c` : champs de l'enregistrement
- chaque champ a son propre type

Définition d'un enregistrement

En dehors des fonctions

```
struct nom_enregistrement {
   type_1 champ_1 ;
      ⋮
   type_n champ_n ;
};
```

Le nom du nouveau type est `struct` *nom_enregistrement*

$\Rightarrow$ Déclaration d'une variable de type enregistrement :

```
struct nom_enregistrement nom_var;
```

Enregistrements et fonctions

Si un enregistrement est passé en paramètre d'une fonction (appel par valeur)
une copie complète est créée

L'utilisation d'un enregistrement comme type de retour permet de retourner plusieurs valeurs

- nécessite une définition de type $\Rightarrow$ lourd
- utile uniquement si champs souvent utilisés ensemble

Alias de type

Utiliser un nouveau nom à la place d'un type

`typedef` *`type existant nouveau_nom`*;

- Aucun nouveau type n'est créé
- Uniquement un raccourci
- On peut ensuite utiliser indifféremment *`type existant`* ou *`nouveau_nom`*

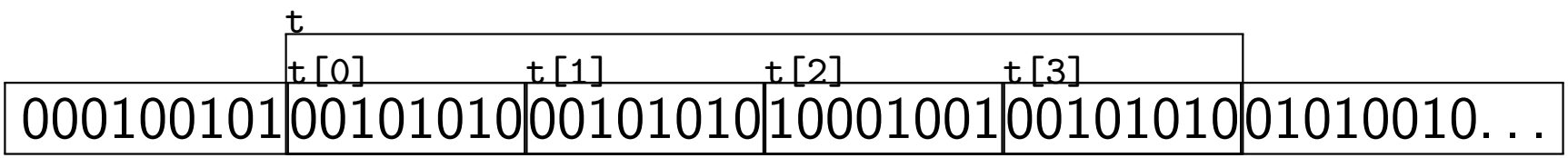
Permet d'éviter d'écrire des `struct` ou des `enum` partout

Tableau

Ensemble fini d'éléments de même type `nom_type`

- Taille fixe (N)
- Application de $[0, N-1]$ dans `nom_type`
- Accès en temps constant via indice

Concrètement : zone de mémoire en cases contiguës



Déclaration statique, accès, écriture

type_elem nom_var[*nb_elems*] ;

nb_elems cases contiguës, chaque case étant interprétée comme un *type_elem*.

Accès à l'élément i

t [i]

$\Rightarrow$ expression (valeur = contenu de la case)

Écriture dans la case i

t [i] = *expr*;

$\Rightarrow$ instruction

Tableaux et fonctions

Type de paramètre :

nom_type nom_param[]

Pas de passage de la taille, si besoin, paramètre supplémentaire

ATTENTION !
Le passage d'un tableau se fait *par référence*

- pas de copie du contenu du tableau
- le contenu du tableau peut être modifié par l'appel

(en fait, valeur d'un tableau = adresse de la première case
`t` = `&(t[0])`)

Accès hors limites

Accès en dehors du tableau : problème

Les bornes du tableau ne sont pas vérifiées

- ni statiquement (à la compilation)
- ni dynamiquement (lors de l'exécution)

- Peut faire planter le programme (erreur de segmentation)
- Ou pire, modifier d'autres variables silencieusement !

Initialisation

Après déclaration, pas initialisé

- contenu = ce qui était là avant en mémoire

Initialisation à la déclaration :

```
type_elem nom_var[nb_elems] =
  { elem_0, ..., elem_k };
```

les éléments après k sont initialisés à 0

Méthode naturelle :

- boucle

Ne jamais utiliser de valeur non initialisée

Invariant de boucle

Expliquer le comportement d'une boucle

Boucle bornée $0 \leq$ `i` $<$ borne

Invariant $=$ proposition logique P

- P vraie pour `i` $= 0$
- à chaque itération, si P vraie et si $i <$ borne,
  alors P vraie après exécution du corps de la boucle

Alors après la boucle :

- P est vraie et $i =$ borne

Invariant de boucle non bornée

Expliquer le comportement d'une boucle

Boucle non bornée condition C

Invariant = proposition logique P

- P vraie en début de boucle
- à chaque itération, si P vraie et si C aussi,
  alors P vraie après exécution du corps de la boucle

Alors après la boucle :

- P est vraie et C est fausse