Définition de types

Ivan Noyer

Crédits

- Ce Wiki de Wikipedia

Crédits

- Ce Wiki de Wikipedia
- Ce cours de OpenClassRoom

Crédits

- Ce Wiki de Wikipedia
- Ce cours de OpenClassRoom
- Ce cours d'Anne Canteaut

Définition

- Une structure est, comme un tableau, un objet de type agrégé, c'est à dire constitué d'un ensemble de valeurs.

Définition

- Une structure est, comme un tableau, un objet de type agrégé, c'est à dire constitué d'un ensemble de valeurs.
- Mais à la différence d'un tableau, ces valeurs peuvent être de types différents.

Définition

- Une structure est, comme un tableau, un objet de type agrégé, c'est à dire constitué d'un ensemble de valeurs.
- Mais à la différence d'un tableau, ces valeurs peuvent être de types différents.
- Par ailleurs, le repérage d'un objet dans la structure se fait, non plus selon un indice, mais par ce qu'on appelle un *nom de champ*, c'est à dire un identificateur.

Définition

- Une structure est, comme un tableau, un objet de type agrégé, c'est à dire constitué d'un ensemble de valeurs.
- Mais à la différence d'un tableau, ces valeurs peuvent être de types différents.
- Par ailleurs, le repérage d'un objet dans la structure se fait, non plus selon un indice, mais par ce qu'on appelle un *nom de champ*, c'est à dire un identificateur.
- On ne peut pas parcourir les champs d'une structure au moyen d'une boucle (alors qu'on le fait pour les tableaux).

Définition

- Une structure est, comme un tableau, un objet de type agrégé, c'est à dire constitué d'un ensemble de valeurs.
- Mais à la différence d'un tableau, ces valeurs peuvent être de types différents.
- Par ailleurs, le repérage d'un objet dans la structure se fait, non plus selon un indice, mais par ce qu'on appelle un *nom de champ*, c'est à dire un identificateur.
- On ne peut pas parcourir les champs d'une structure au moyen d'une boucle (alors qu'on le fait pour les tableaux).
- Une *structure* est donc une suite finie d'objets de type différents (les *champs* ou *membres*). Le nombre de ces différents champs est limité à 127 soit $2^7 - 1$.

Exemple d'utilisation

Une structure permet d'accroître la clarté des programmes en rassemblant dans un même objet des informations possédant un lien. Par exemple :

- Les différentes informations associées à une personne : nom, prénom, adresse, sexe, numéro de SS...

Exemple d'utilisation

Une structure permet d'accroître la clarté des programmes en rassemblant dans un même objet des informations possédant un lien. Par exemple :

- Les différentes informations associées à une personne : nom, prénom, adresse, sexe, numéro de SS...
- les différentes informations associées à un point du plan : ses coordonnées, et, s'il s'agit d'un pixel, ses composantes RVB.

Un mot sur les unions

(Hors programme)

- Une *union* est un moyen artificiel de considérer un même objet avec différents types.

Un mot sur les unions

(Hors programme)

- Une *union* est un moyen artificiel de considérer un même objet avec différents types.
- A chaque type se trouve associé, comme pour les structures, un nom de champs.

Un mot sur les unions

(Hors programme)

- Une *union* est un moyen artificiel de considérer un même objet avec différents types.
- A chaque type se trouve associé, comme pour les structures, un nom de champs.
- Mais dans une structure, les champs ne s'excluent pas l'un, l'autre : ils sont juxtaposés. Dans une union, les champs ne peuvent cooexister en même temps : ils sont superposés.

Un mot sur les unions

(Hors programme)

- Une *union* est un moyen artificiel de considérer un même objet avec différents types.
- A chaque type se trouve associé, comme pour les structures, un nom de champs.
- Mais dans une structure, les champs ne s'excluent pas l'un, l'autre : ils sont juxtaposés. Dans une union, les champs ne peuvent cooexister en même temps : ils sont superposés.
- Les unions s'utilisent

Un mot sur les unions

(Hors programme)

- Une *union* est un moyen artificiel de considérer un même objet avec différents types.
- A chaque type se trouve associé, comme pour les structures, un nom de champs.
- Mais dans une structure, les champs ne s'excluent pas l'un, l'autre : ils sont juxtaposés. Dans une union, les champs ne peuvent cooexister en même temps : ils sont superposés.
- Les unions s'utilisent
  - pour interpréter de façons différentes un même motif binaire.

Un mot sur les unions

(Hors programme)

- Une *union* est un moyen artificiel de considérer un même objet avec différents types.
- A chaque type se trouve associé, comme pour les structures, un nom de champs.
- Mais dans une structure, les champs ne s'excluent pas l'un, l'autre : ils sont juxtaposés. Dans une union, les champs ne peuvent cooexister en même temps : ils sont superposés.
- Les unions s'utilisent
  - pour interpréter de façons différentes un même motif binaire.
  - pour économiser de la mémoire en faisant occuper à des instants différents un même emplacement par des informations de types différents.

Un mot sur les unions

(Hors programme)

- Une *union* est un moyen artificiel de considérer un même objet avec différents types.
- A chaque type se trouve associé, comme pour les structures, un nom de champs.
- Mais dans une structure, les champs ne s'excluent pas l'un, l'autre : ils sont juxtaposés. Dans une union, les champs ne peuvent cooexister en même temps : ils sont superposés.
- Les unions s'utilisent
  - pour interpréter de façons différentes un même motif binaire.
  - pour économiser de la mémoire en faisant occuper à des instants différents un même emplacement par des informations de types différents.
- L'union a un caractère peu portable et reste limitée à des utilisations particulières.

Définir de nouveaux types

Les unions et structures peuvent être considérés comme des types définis par le programmeur.

Modéle et déclaration

- Un *modèle de structure* est la description d'une structure :

```
1 struct modele
2 {type_1 membre_1;
3   type_2 membre_2;
4     ...
5   type_n membre_n;
6 };// ce point virgule est impératif !!
```

Modéle et déclaration

- Un *modèle de structure* est la description d'une structure :

```
1 struct modele
2 {type_1 membre_1;
3   type_2 membre_2;
4     ...
5   type_n membre_n;
6 };// ce point virgule est impératif !!
```

- Une fois le modèle déclaré, on déclare un objet de ce type avec :

```
1 struct modele objet  ;
```

Modéle et déclaration

- Un *modèle de structure* est la description d'une structure :

```
struct modele
{type_1 membre_1;
  type_2 membre_2;
    ...
  type_n membre_n;
};// ce point virgule est impératif !!
```

- Une fois le modèle déclaré, on déclare un objet de ce type avec :

```
struct modele objet  ;
```

- Cette syntaxe qui déclare dans la foulée le type `modele` et un objet `representant` de ce type est également possible :

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

Exemple

- On définit une structure `temps` qui modélise un triplet (heure,minute,secondes) :

```
1 struct temps {
2     unsigned int heures;
3     unsigned int minutes;
4     double secondes;
5 };// ce point virgule est impératif !!
```

Exemple

- On définit une structure `temps` qui modélise un triplet (heure,minute,secondes) :

```
1 struct temps {
2     unsigned int heures;
3     unsigned int minutes;
4     double secondes;
5 };// ce point virgule est impératif !!
```

- La déclaration d'un objet de type `temps` se fait ainsi :

```
1 struct temps t;
```

Affectation

Il reste à affecter les différents champs d'un objet `obj` de type structuré `modele`. Il y a deux possibilités :

Syntaxe de type tableau `struct modele obj = {v1, ..., vn }` pour une déclaration/affectation simultanée.
Les types de $v_1, \ldots, v_n$ sont ceux des champs de `modele` et respecte l'ordre des champs dans la structure.

Affectation

Il reste à affecter les différents champs d'un objet `obj` de type structuré `modele`. Il y a deux possibilités :

Syntaxe de type tableau `struct modele obj = {v1, ..., vn }` pour une déclaration/affectation simultanée.
Les types de $v_1, \ldots, v_n$ sont ceux des champs de `modele` et respecte l'ordre des champs dans la structure.

Syntaxe pointée `obj.membre_1 = v1` ,..., `obj.membre_n = vn` .

Exemple

- Affichage d'une structure de temps :

```
1 void affichage(struct temps t){
2     printf("temps=%uh %umin %fs\n",t.heures,t.minutes,t.
      secondes);
3 }
4
```

Exemple

- Affichage d'une structure de temps :

```
1 void affichage(struct temps t){
2     printf("temps=%uh %umin %fs\n",t.heures,t.minutes,t.
    secondes);
3 }
4
```

- Affectation

```
1 void main() {
2   struct temps t1 = { 1, 45, 30.560 };//Syntaxe type
      tableau
3   affiche(t1);
4   struct temps t2;
5   t2.heures = 5;//Syntaxe de type pointée
6   t2.minutes = 23;
7   t2.secondes = 10.06;
8   affiche(t2);
9 }
10
```

Exemple

- Affectation

```
1  void main() {
2    struct temps t1 = { 1, 45, 30.560 };//Syntaxe type
       tableau
3    affiche(t1);
4    struct temps t2;
5    t2.heures = 5;Syntaxe de type pointée
6    t2.minutes = 23;
7    t2.secondes = 10.06;
8    affiche(t2);
9  }
10
```

Exemple

- Affectation

```
1  void main() {
2    struct temps t1 = { 1, 45, 30.560 };//Syntaxe type
       tableau
3    affiche(t1);
4    struct temps t2;
5    t2.heures = 5;Syntaxe de type pointée
6    t2.minutes = 23;
7    t2.secondes = 10.06;
8    affiche(t2);
9  }
10
```

- On obtient :

```
temps=1h 45min 30.560000s
temps=5h 23min 10.060000s
```

Alias de structure

- L'écriture `struct temps t` est un peu lourde. Il est plus agréable de faire des déclarations sous la forme `temps t`.

Alias de structure

- L'écriture `struct temps t` est un peu lourde. Il est plus agréable de faire des déclarations sous la forme `temps t`.
- Pour celà, on utilise l'instruction `typedef` dont voici la syntaxe :

```
1 typedef struct nomModele nomAlias;
2 struct nomModele{
3   // les différents champs et leurs types
4 };
```

Alias de structure

- L'écriture `struct temps t` est un peu lourde. Il est plus agréable de faire des déclarations sous la forme `temps t`.
- Pour celà, on utilise l'instruction `typedef` dont voici la syntaxe :

```
1 typedef struct nomModele nomAlias;
2 struct nomModele{
3   // les différents champs et leurs types
4 };
```

- `typedef` indique que nous créons un alias de structure, `struct nomModele` est le nom de la structure et `nomAlias` celui de son alias

Exemple

```
1 typedef struct temps time;
2 struct temps {
3     unsigned int heures;
4     unsigned int minutes;
5     double secondes;
6 };// ce point virgule est impératif !!
7
```

Exemple

```
1 typedef struct temps time;
2 struct temps {
3     unsigned int heures;
4     unsigned int minutes;
5     double secondes;
6 };// ce point virgule est impératif !!
7
```

- Fonction d'affichage

```
1 void affiche(time t){
2     printf("temps=%ih %imin %fs\n",t.heures,t.minutes,t.
      secondes);
3 }
4
```

Exemple

Affectation

```
1 void main() {
2   time t2;
3   t2.heures = 5;
4   t2.minutes = 23;
5   t2.secondes = 10.06;
6   affiche(t2);
7 }
8
```

Pointeur de structure

- Déclaration sans surprise d'un pointeur sur notre type structuré `temps` :

```
1 time t;
2 time* p  = &t;// ou time *p = &t
```

Pointeur de structure

- Déclaration sans surprise d'un pointeur sur notre type structuré `temps` :

```
time t;
time* p  = &t;// ou time *p = &t
```

- Il y a en revanche un peu de sucre syntaxique ; on peut affecter les champs de l'objet sur lequel pointe le pointeur de deux façons :

```
(*p1).heures = 12;// façon classique, noter la parenthèse
p1->minutes = 41;// façon plus élégante
p1->secondes = 23.56;
```

Pointeur de structure

- Déclaration sans surprise d'un pointeur sur notre type structuré `temps` :

```
time t;
time* p  = &t;// ou time *p = &t
```

- Il y a en revanche un peu de sucre syntaxique ; on peut affecter les champs de l'objet sur lequel pointe le pointeur de deux façons :

```
(*p1).heures = 12;// façon classique, noter la parenthèse
p1->minutes = 41;// façon plus élégante
p1->secondes = 23.56;
```

- `*p1.heures = 15;` produit une erreur de compilation.

Initialisation sélective

- On peut recourir à une *initialisation sélective* en nommant bien les champs. Dans ce cas, l'ordre des membres n'a pas d'importance (comme pour le passage des paramètres nommés en `Python`) :

```
1 time t = { .secondes = 30.560, .minutes = 45,
2     .heures = 1 };
```

Initialisation sélective

- On peut recourir à une *initialisation sélective* en nommant bien les champs. Dans ce cas, l'ordre des membres n'a pas d'importance (comme pour le passage des paramètres nommés en `Python`) :

```
time t = { .secondes = 30.560, .minutes = 45,
    .heures = 1 };
```

- On peut aussi mélanger les plaisirs :

```
time t = { .minutes = 45, 30.560 };
```

Initialisation sélective

- On peut recourir à une *initialisation sélective* en nommant bien les champs. Dans ce cas, l'ordre des membres n'a pas d'importance (comme pour le passage des paramètres nommés en `Python`) :

```
time t = { .secondes = 30.560, .minutes = 45,
    .heures = 1 };
```

- On peut aussi mélanger les plaisirs :

```
time t = { .minutes = 45, 30.560 };
```

- Il manque la valeur pour les heures : par défaut, elle vaut 0. Ensuite on a une initialisation sélective des minutes, puis on revient à l'initialisation séquentielle pour les secondes.

Initialisation sélective

- On peut recourir à une *initialisation sélective* en nommant bien les champs. Dans ce cas, l'ordre des membres n'a pas d'importance (comme pour le passage des paramètres nommés en `Python`) :

```
time t = { .secondes = 30.560, .minutes = 45,
    .heures = 1 };
```

- On peut aussi mélanger les plaisirs :

```
time t = { .minutes = 45, 30.560 };
```

- Il manque la valeur pour les heures : par défaut, elle vaut 0. Ensuite on a une initialisation sélective des minutes, puis on revient à l'initialisation séquentielle pour les secondes.
- L'initialisation séquentielle reprend automatiquement à la dernière initialisation sélective.

Initialisation sélective

- On peut recourir à une *initialisation sélective* en nommant bien les champs. Dans ce cas, l'ordre des membres n'a pas d'importance (comme pour le passage des paramètres nommés en `Python`) :

```
time t = { .secondes = 30.560, .minutes = 45,
    .heures = 1 };
```

- On peut aussi mélanger les plaisirs :

```
time t = { .minutes = 45, 30.560 };
```

- Il manque la valeur pour les heures : par défaut, elle vaut 0. Ensuite on a une initialisation sélective des minutes, puis on revient à l'initialisation séquentielle pour les secondes.
- L'initialisation séquentielle reprend automatiquement à la dernière initialisation sélective.
- A retenir : Si certains membres d'une structure ne se voient pas attribuer de valeurs durant l'initialisation d'une variable, ils seront initialisés à zéro ou, s'il s'agit de pointeurs, seront des pointeurs nuls.

Structure et mémoire (Hors programme)

Les différents champs d'une structure sont contigus en mémoire.

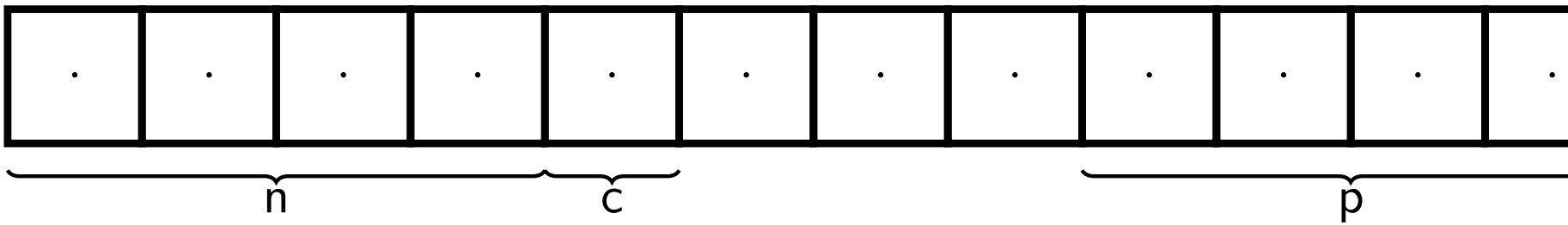
- Dans certaines implémentations, les `int` sont alignés sur des adresses multiples de 4 (leur premier octet est nécessairement sur une adresse multiple de 4).

Structure et mémoire (Hors programme)

Les différents champs d'une structure sont contigus en mémoire.

- Dans certaines implémentations, les `int` sont alignés sur des adresses multiples de 4 (leur premier octet est nécessairement sur une adresse multiple de 4).
- Considérons, dans une telle implémentation, la déclaration

```
1 struct essai{ int n;// 4 octets
2               char c;// 1 octet
3               int p; // 4 octets
4       };// voir ci-dessous le stockage en mémoire :
5
```

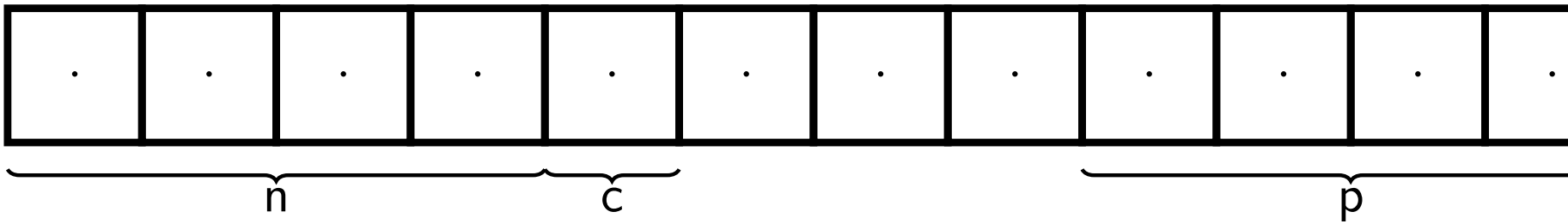


Structure et mémoire (Hors programme)

Les différents champs d'une structure sont contigus en mémoire.

- Dans certaines implémentations, les `int` sont alignés sur des adresses multiples de 4 (leur premier octet est nécessairement sur une adresse multiple de 4).
- Considérons, dans une telle implémentation, la déclaration

```
1 struct essai{ int n;// 4 octets
2               char c;// 1 octet
3               int p; // 4 octets
4       };// voir ci-dessous le stockage en mémoire :
5
```



- On remarque l'utilisation d'*octets de remplissage* entre les champs `.c` et `.p`. Donc la taille de la structure est 12 octets et non 9.

Structures et tableaux de structures

Supposons que notre implémentation impose aux `int` de commencer à des adresses multiples de 4.

- Rappel : Les différents champs d'une structure sont contigus en mémoire.

Structures et tableaux de structures

Supposons que notre implémentation impose aux `int` de commencer à des adresses multiples de 4.

- Rappel : Les différents champs d'une structure sont contigus en mémoire.
- Considérons

```
1 struct essai{ int n;// 4 octets
2               char c;// 1 octet
3         };
4
```

Structures et tableaux de structures

Supposons que notre implémentation impose aux `int` de commencer à des adresses multiples de 4.

- Rappel : Les différents champs d'une structure sont contigus en mémoire.
- Considérons

```
1 struct essai{ int n;// 4 octets
2               char c;// 1 octet
3       };
4
```

- En théorie, 5 octets suffisent pour représenter un objet de type `essai`. Mais un tel objet peut se retrouver dans un tableau...

Structures et tableaux de structures

Supposons que notre implémentation impose aux `int` de commencer à des adresses multiples de 4.

- Rappel : Les différents champs d'une structure sont contigus en mémoire.
- Considérons

```
1 struct essai{ int n;// 4 octets
2               char c;// 1 octet
3       };
4
```

- En théorie, 5 octets suffisent pour représenter un objet de type `essai`. Mais un tel objet peut se retrouver dans un tableau...
- La règle de contguïté entre deux éléments successifs d'un tableau impose la violation de la règle des adresses en multiple de 4.

Structures et tableaux de structures

Supposons que notre implémentation impose aux `int` de commencer à des adresses multiples de 4.

- Rappel : Les différents champs d'une structure sont contigus en mémoire.
- Considérons

```
1 struct essai{ int n;// 4 octets
2               char c;// 1 octet
3       };
4
```

- En théorie, 5 octets suffisent pour représenter un objet de type `essai`. Mais un tel objet peut se retrouver dans un tableau...
- La règle de contguïté entre deux éléments successifs d'un tableau impose la violation de la règle des adresses en multiple de 4.
- Pour cette raison, les objets de type `essai` font 4+1+3 soit 8 octets DANS cette implémentation.

Structure et mémoire ♡

- On le voit, la taille des objets structurés dépend de l'implémentation.

Structure et mémoire ♡

- On le voit, la taille des objets structurés dépend de l'implémentation.
- Pour se simplifier la vie : on considère en CPGE que la taille d'un type structuré est la somme des tailles de ses différents champs.

Affectation et retour par copie

- Affectation par copie :

```
1 struct S s= {...}; //déclaration et initialisation de s
2 struct S t; // t non initialisée
3 t = s; // t reçoit une copie de s
4
```

Affectation et retour par copie

- Affectation par copie :

```
1 struct S s= {...}; //déclaration et initialisation de s
2 struct S t; // t non initialisée
3 t = s; // t reçoit une copie de s
4
```

- Retour d'une valeur structurée :

```
1 struct S f(...){
2   ...
3   struct S t = ...;// t est initialisée
4   return t; // t est copié dans l'emplacement
5             // qui accueille le retour de la fonction
6         }
7
```

Un mot sur le passage de structures en paramètres

- En C, le passage des arguments s'effectue par *valeur*. Cela signifie que la fonction travaille avec une copie de l'argument. Si l'argument est de grande taille, on utilise donc beaucoup de mémoire pour stocker une information redondante.

Un mot sur le passage de structures en paramètres

- En C, le passage des arguments s'effectue par *valeur*. Cela signifie que la fonction travaille avec une copie de l'argument. Si l'argument est de grande taille, on utilise donc beaucoup de mémoire pour stocker une information redondante.
- Il est préferable de passer en paramètre des fonctions, non pas une structure, mais un pointeur sur cette structure. Ce pointeur sert de « poignée » à l'objet et sa taille est fixe quelle que soit la structure pointée.

Un mot sur le passage de structures en paramètres

- En C, le passage des arguments s'effectue par *valeur*. Cela signifie que la fonction travaille avec une copie de l'argument. Si l'argument est de grande taille, on utilise donc beaucoup de mémoire pour stocker une information redondante.
- Il est préferable de passer en paramètre des fonctions, non pas une structure, mais un pointeur sur cette structure. Ce pointeur sert de « poignée » à l'objet et sa taille est fixe quelle que soit la structure pointée.
- Dans le même soucis de réduire le trafic dans la mémoire, on ne renvoie pas une structure mais, de préférence, un pointeur sur cette structure.

Déclaration anticipée

- La norme autorise qu'on déclare un nom de structure sans sa description :

```
1 struct enreg; // enreg est un nom de structure
2               // sa description sera donnée plus tard
3
```

Déclaration anticipée

- La norme autorise qu'on déclare un nom de structure sans sa description :

```
1 struct enreg; // enreg est un nom de structure
2               // sa description sera donnée plus tard
3
```

- Dans la portée de l'identificateur de structure, il est permis de fournir plus tard la description du type :

```
1 struct enreg;// déclaration du type
2 ...
3 struct enreg {char c; int n;}; // description du type
4
```

Dépendance mutuelle

- La déclaration anticipée est indispensable en cas de dépendance mutuelle entre 2 structures.

Dépendance mutuelle

- La déclaration anticipée est indispensable en cas de dépendance mutuelle entre 2 structures.
- Exemple :

```
1  struct machin; // déclaration anticipée
2  struct chose{... // definition normale de chose
3       ...
4       struct machin *adm;// pointeur sur objet type machin
5    };
6  struct machin{...// def. tardive de machin
7    ...
8    struct chose *adc;// pointeur sur objet type chose
9    }
10
```

Fonctions mutuellement récursives

Il n'y a pas que pour les dépendances mutuelles entre structures que la déclaration anticipée est indispensable.

- La déclaration anticipée est indispensable en cas de récursion mutuelle entre deux fonctions.

Fonctions mutuellement récursives

Il n'y a pas que pour les dépendances mutuelles entre structures que la déclaration anticipée est indispensable.

- La déclaration anticipée est indispensable en cas de récursion mutuelle entre deux fonctions.
- Exemple : Ecrire deux fonctions mutuellement récursives `bool pair(unsigned)` et `bool impair(unsigned)`.

Fonctions mutuellement récursives

Il n'y a pas que pour les dépendances mutuelles entre structures que la déclaration anticipée est indispensable.

- La déclaration anticipée est indispensable en cas de récursion mutuelle entre deux fonctions.
- Exemple : Ecrire deux fonctions mutuellement récursives `bool pair(unsigned)` et `bool impair(unsigned)`.
- Il faut vraiment que je place ce transparent aiileurs que dans le cours sur les structures !

Présentation

Il s'agit de définir un type par la liste exhaustive de ses habitants. Par exemple :

```
1 enum couleur = {rouge, vert, bleu};
```

En fait, les valeurs représentées sont des entiers. Vérifions le avec :

```
1 enum couleur  {rouge, vert, bleu};
2
3 void main(){
4   enum couleur c;
5   c = bleu;
6   printf("c = %d\n",c);
7 }
```

Après compilation/exécution :

```
c = 2
```

On comprend que `rouge` est représenté par défaut par 0, `vert` par 1 et `bleu` par 2.

Modifier les valeurs par défaut

Il est possible de modifier les valeurs par défaut lors de la déclaration :

```
1    enum couleur {rouge=11, vert=17, bleu=23};
2
```

Alors, la séquence :

```
1    enum couleur c; c = bleu;
2    printf("c = %d\n",c);
```

, affiche

```
c=23
```