Type énuméré ; mot clé `enum`

1. Introduction

Supposons que vous ayez besoin d'une variable dont la valeur représente :

- un jour de la semaine
- une des 12 notes de musique
- un des types de lexème d'un langage de programmation (symbole, nombre, opérateur…)
- …

… c'est à dire **une variable dont la valeur est l'un des éléments d'une énumération de valeurs exclusives possibles.**
Quel type utiliser pour déclarer cette variable ?

Une solution est d'utiliser un type entier (un int, par exemple).

Par exemple, pour travailler avec les jours de la semaine, on peut coder lundi par la valeur 0, mardi par la valeur 1, …., et dimanche par la valeur 6 :

```
int main ( int argc , char * argv[] ) {
  int jourSemaine= 1 ; // jourSemaine vaut 1 <=> représente mardi
  if(jourSemaine == 0) {
          printf("lundi, c'est ravioli\n") ;
  }
  …
  return 0 ;
}
```

Cette solution serait avantageusement complétée en définissant des constantes nommées :

```
#define LUNDI    0
#define MARDI    1
#define MERCREDI         2
#define JEUDI    3
#define VENDREDI         4
#define SAMEDI   5
#define DIMANCHE         6
int main ( int argc , char * argv[] ) {
  int jourSemaine = MARDI ;      // jourSemaine représente mardi
                                                  // et vaut 2
  if(jourSemaine == DIMANCHE) {
          printf("c'est dimanche. Au boulot sur le projet info !\n") ;
  }
  …
  return 0 ;
}
```

C'est déjà mieux, car on utilise des identificateurs clairs LUNDI, MARDI, etc dans le code, à la place de nombres entiers qui fort peu signifiants.

Mais ce n'est pas parfait. Par exemple, un petit malin pourra écrire :

```
int jourSemaine = 255 ;      // euh… c'est quoi le jour 255 ?
```

En fait, un « jour de la semaine » ne devrait pouvoir avoir pour valeur que LUNDI, …, DIMANCHE, à l'exclusion de tout autre valeur. Le type donc pas int n'est pas adapté.

Les types énumérés sont la bonne solution pour ce genre de situation.

2. Type énuméré

En C, le mot clé enum permet de déclarer un type énuméré. On liste ensuite dans la déclaration du type les valeurs (constantes symboliques, ou « étiquettes ») possibles pour ce type.

```
enum jour {
  LUNDI,
  MARDI,
  MERCREDI,
  JEUDI,
  VENDREDI,
  SAMEDI,
  DIMANCHE
} ;
```

Ce qui précède déclare un nouveau type énuméré, dont le nom est enum jour, ainsi que des étiquettes LUNDI, MARDI, … DIMANCHE associées à ce type enum jour.

La chose importante avec les types énumérés est qu'une variable de type énuméré ne peut prendre pour valeur que l'une des étiquettes associées au type.

Ainsi, une variable de type enum jour ne peut prendre pour valeur que LUNDI, MARDI, … ou DIMANCHE, à l'exclusion de toute autre valeur.

On peut donc écrire :

```
int main ( int argc , char * argv[] ) {
   enum jour jourSemaine = LUNDI ;
         // jourSemaine est une variable de type enum jour
   if(jourSemaine == DIMANCHE) {
        printf("c'est dimanche. Grasse mat' ! \n") ;
   }
  …
   return 0 ;
}
```

Mais par contre, il n'est plus possible d'écrire :

```
enum jour jourSemaine = 1 ; // INTERDIT
// => on est forcé d'utiliser les étiquettes
// Et c'est une bonne chose pour la clarté du code !
```

Et encore moins :

```
enum jour jourSemaine = 255 ; // INTERDIT (heureusement...)
```

3. Valeur entière des étiquettes d'un type énuméré

Dans votre dos, le compilateur C associe automatiquement des valeurs entières à chaque étiquette du type énuméré.

Ainsi, dans l'exemple précédent, l'identificateur LUNDI vaut 0, MARDI vaut 2, … et DIMANCHE vaut 6. On peut le voir avec le code suivant

```
enum jour jourSemaine = MARDI ;
printf("Valeur en entier de jourSemaine : %d\n", jourSemaine) ;
// afficherait dans la Terminal :
//               Valeur en entier de jourSemaine : 1
```

Il est possible de forcer les valeurs associées à chaque étiquette :

```
enum volumesonore {
  AUCUN = 0,
  FAIBLE = 20,
  MOYEN = 50,
  FORT = 100
} ;
```

4. Quelques compléments autour des types énumérés

Comme avec les types structurés (struct), on utilise souvent typedef pour renommer un type énuméré :

```
enum notemusique { DO, RE, MI, FA, SOL, LA, SI, DO} ;
typedef enum notemusique note_t ;
// note_t est désormais le même type que enum notemusique
```

Et bien sûr, comme avec les types structurés, on peut utiliser typedef dès la déclaration du type énuméré :

```
typedef enum { DO, RE, MI, FA, SOL, LA, SI, DO} note_t ;
// déclare directement le type énuméré note_t
```

Pour pouvoir afficher dans le Terminal une chaine de caractère signifiante (au lieu de valeurs entières incompréhensible…), on a souvent recours à une fonction réalisant la conversion d'une variable énumérée vers une chaine de caractères :

```
typedef enum {AUCUN, FAIBLE, MOYEN, FORT } niveau_t ;

char * niveauToStr(niveau_t n) {
  switch(n) {
          case AUCUN :
                  return "pas de son" ;
          case FAIBLE :
                  return "faible" ;
          case MOYEN :
                  return "moyen" ;
          case FORT :
                  return "max" ;
  }
}
```

```c
int main ( int argc , char * argv[] ) {
  niveau_t level = FORT ;
  printf("le niveau est %s\n", niveauToStr(level) ;
       // affiche « le niveau est max »
  return 0 ;
}
```

A l'inverse, comme on ne peut pas directement affecter un entier à une variable de type énuméré, on a parfois besoin d'écrire des fonctions de conversion de type int → enum (qui, au passage, vont vérifier que la valeur entière considérée est bien l'une des valeurs valides). Exemple :

```c
niveau_t intToNiveau(int valeur) {
    switch(n) {
    case AUCUN :
        return AUCUN;
    case FAIBLE :
        return FAIBLE;
    case MOYEN :
        return MOYEN;
    case FORT :
        return FORT;
    default :
        // dans tous les autres cas, c'est absurde
        printf("Erreur, valeur invalide\n") ;
        exit(1) ; // on arrête le programme
    }
}
```

Et parfois, on écrit aussi des fonctions de conversion depuis une chaine de caractères :

```c
niveau_t strToNiveau(char * chaine) {
    if(strcmp(chaine, "pas de son") == 0) {
        return AUCUN;
    } else if(strcmp(chaine, "faible") == 0) {
        return FAIBLE;
    } else if(strcmp(chaine, "moyen") == 0) {
        return MOYEN;
    } else if(strcmp(chaine, "max") == 0) {
        return FORT;
    } else {
        printf("Erreur, chaine non reconnue\n") ;
        exit(1) ; // on arrête le programme
    }
}
```

5. Exemple

Dans l'exemple suivant, pour codifier les diverses erreurs possibles d'un programme, on déclare et utilise un type énuméré. Puis, chaque erreur possible est associée à une chaine de caractère pour afficher en clair l'erreur dans le Terminal.

```
typedef enum {
    OK = -1,
    ERREUR_OUVERTURE,    // vaudra 0
    ERREUR_FORMAT        // vaudra 1
} error_t ;


// on stocke les chaines correspondante dans un tableau
// ... on utilise donc le fait que FICHIER_INCONNU vaut 0
// indice de la première case du tableau. Ruse !
const char * error_string [] = {
    "Le fichier n'a pas pu être ouvert",
    "Le format de fichier est incorrect"
} ;



// Lecture d'un fichier texte contenant un unique entier
// Retourne un code erreur error_t :
// OK si tout va bien
// Un autre valeur en cas d'erreur
error_t lireFichier(char *nom, int *pValeurLue) {
    FILE * f = fopen(nom, "r") ;
    if(f == NULL) {
        return ERREUR_OUVERTURE ;
    }
    if(fscanf(f, "%d", pValeurLue) == 0) {
        return ERREUR_FORMAT ;
    }
    return OK ;
}



main() {
    int x ;
    error_t errCode = lireFichier("toto.txt", &x) ;
    if(err != OK) {
        printf("ERREUR ==> %s\n",  error_string[errCode] ) ;
        exit(1) ;
    }
    return 0 ;
}
```