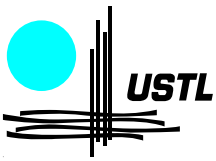

Licence d'informatique
Module de Pratique du C

Travaux dirigés

Manipulation de structures

Philippe MARQUET

Octobre 2004

Les exercices proposés illustrent l'utilisation simple de structures. Une structure, ou enregistrement, est un agrégat de données. Une structure définit un nouveau type de donnée dont les éléments, ou champs, sont éventuellement hétérogènes (de types différents). Ces champs sont accessibles par leur nom.

On traite aussi d'unions et de types énumérés.

Nous reviendrons sur des utilisations plus avancées des structures et de leur représentation en mémoire dans d'autres TD.

Exercice 1 (Jouons aux cartes)
On désire gérer des cartes à jouer.

- On joue au bridge avec un jeu de 4 (trèfle, carreau, coeur, pique) $\times$ 13 (as à 10 + valet, dame, roi) = 52 cartes.
- Au tarot il faut aussi considérer un cavalier dans chaque couleur, les atouts (de 1 à 21), et l'excuse.

Question 1.1 Proposez des structures de données pour représenter une carte d'un jeu de bridge, puis une carte d'un jeu de tarot.
(• ***Éléments de solution 1.1***

- On débute par un une définition des couleurs. On peut écrire

```
#define TREFFLE   0
#define CARREAU   1
#define COEUR     2
#define PIQUE     3
```

  pour assurer l'unicité des valeurs, on peut choisir de définir

```
#define TREFFLE   0
#define CARREAU   (TREFFLE+1)
#define COEUR     (CARREAU+1)
#define PIQUE     (COEUR+1)
```

  mais il est préférable d'utiliser

```
enum couleur_e {TREFFLE, CARREAU, COEUR, PIQUE};
```

  qui définit les quatre valeurs constantes entières `TREFFLE`, `CARREAU`, `COEUR`, et `PIQUE` et un type `enum couleur_e`.
- La valeur d'une carte peut être représentée par un entier. Les valeurs de 1 à 10 pour les cartes, 11 pour le valet, 12 la dame et 13 le roi. On peut par exemple écrire :

```
#define VALET 11
#define DAME  12
#define ROI   13
```

  ou préférer

```
enum tete_e {VALET=11, DAME, ROI};
```

  ou même un `enum` anonyme :

```
enum {VALET=11, DAME, ROI};
```

ne définissant rien d'autre que les trois valeurs constantes.

– On en vient à la définition d'un type pour associer les informations relatives à une carte :

```
struct carte_bridge_s {
    enum couleur_e cb_couleur;
    int cb_valeur;
};
```

on peut remarquer le nommage des champs, une habitude personnelle.

– On peut définir des variables comme suit :

```
struct carte_bridge_s une_carte;
struct carte_bridge_s une_main[10];
```

– S'il est possible de mixer la définition du type et la déclaration des variables en faisant

```
struct carte_bridge_s {
    enum couleur_e cb_couleur;
    int cb_valeur;
} une_carte, une_main[10];
```

c'est une mauvaise pratique.

– Pour le jeu de tarot

```
enum couleur_e {TREFFLE, CARREAU, COEUR, PIQUE, ATOUT, EXCUSE};
enum {VALET=11, CAVALIER, DAME, ROI};
struct carte_tarot_s {
    enum couleur_e ct_couleur;
    int ct_valeur;
};
```

•)

Question 1.2 Au tarot, on identifie le 1 et le 21 d'atout ainsi que l'excuse comme des bouts. Proposez une fonction `est_un_bout` qui retourne vrai si et seulement si la valeur d'une carte passée en paramètre est un bout.

(• ***Éléments de solution 1.2***

– On commence par le prototype (passage d'une valeur de type structure en paramètre) :

```
int est_un_bout(struct carte_tarot_s);
```

ou (et je préfère...)

```
int est_un_bout(struct carte_tarot_s carte);
```

– On ne peut comparer directement deux valeurs de structures avec l'opérateur `==`. On écrit donc une fonction de comparaison de deux cartes :

```
int est_egal_tarot(struct carte_tarot_s ca, struct carte_tarot_s cb)
{
    return ca.ct_valeur == cb.ct_valeur
        && ca.ct_couleur == cb.ct_couleur;
}
```

on remarque le moyen d'accéder à un champs, c'est ma première fois...

– Comme définition de la fonction `est_un_bout`, on peut bien entendu écrire directement

```
int
est_un_bout(struct carte_tarot_s carte)
{
    return (carte.ct_couleur == EXCUSE && carte.ct_valeur == 0)
        || (carte.ct_couleur == ATOUT && carte.ct_valeur == 1)
        || (carte.ct_couleur == ATOUT && carte.ct_valeur == 21);
}
```

on peut préférer définir des valeurs pour les trois bouts :

```
int
est_un_bout(struct carte_tarot_s carte)
{
    const struct carte_tarot_s
        BOUT_E = {EXCUSE, 0},
        BOUT_PETIT = {ATOUT, 1},
        BOUT_21 = {ATOUT, 21};

    return est_egal_tarot(carte, BOUT_E)
        || est_egal_tarot(carte, BOUT_PETIT)
        || est_egal_tarot(carte, BOUT_21);
}
```

on remarque les initialisations des variables (constantes, mais peu importe) `BOUT_*` par des valeurs littérales. Cependant, on ne peut écrire d'affectation avec de telles valeurs :

```
struct carte_tarot_s c;

c = {EXCUSE, 0};                /* INTERDIT */
```

pas plus que des utilisations de telles valeurs en paramètre :

```
est_egal(carte, {EXCUSE, 0})    /* INTERDIT */
```

•)

La belote se joue à quatre joueurs avec un jeu de 32 cartes (on exclut les cartes 2 à 6). Une des couleurs est désignée comme atout. L'ordre des cartes est le suivant :

- pour l'atout : valet, 9, as, 10, roi, dame, 8, 7 ;
- pour les autres couleurs : as, 10, roi, dame, valet, 9, 8, 7.

À chaque tour de jeu, un premier joueur joue la carte de son choix. Cette carte indique la couleur demandée. Chacun des autres joueurs joue une carte. La levée est emportée par le plus gros atout, ou, s'il n'y en a pas, pas la plus forte carte de la couleur demandée.

Question 1.3 Proposez une fonction qui retourne la carte emportant la levée identifiée par les paramètres : quatre cartes, l'atout, la couleur demandée.

(• ***Éléments de solution 1.3***

- Quatre cartes... un tableau, on a donc un prototype comme

```
struct carte_belote_s gagne_la_levee(struct carte_belote_s levee[], /* 4 cartes */
                                     enum couleur_b_e atout,
                                     enum couleur_b_e demande);
```

  on retourne une carte, c'est le sujet : on peut retourner des valeurs qui sont des structures.
- On peut écrire

```
struct carte_belote_s gagne_la_levee(struct carte_belote_s levee[4],
                                     enum couleur_b_e atout,
                                     enum couleur_b_e demande);
```

  mais le 4 étant ignoré, je préfère ne pas le mettre et laisser mon commentaire.
- Et une proposition pour le code de la fonction :

```
/* Comparaison de valeurs, retournent vrai ssi val_carte1 > val_carte2) */
int plus_fort_atout(int val_carte1, int val_carte2);
int plus_fort(int val_carte1, int val_carte2);

struct carte_belote_s
gagne_la_levee(struct carte_belote_s levee[], /* 4 cartes */
               enum couleur_b_e atout,
               enum couleur_b_e demande)
{
    int i;
#define NONE -1
    int max_atout = NONE;       /* indice du meilleur atout */
```

```
    int max = NONE;             /* indice du meilleur non atout */

    /* On regarde les atouts */
    for (i=0 ; i<4 ; i++)
        if (levee[i].cb_couleur == atout)
            if (max_atout == NONE)
                max_atout = i;
            else if (plus_fort_atout(levee[i].cb_valeur,
                                     levee[max_atout].cb_valeur))
                max_atout = i;

    if (max_atout != NONE)
        return levee[max_atout];

    /* On regarde la couleur demandee */
    for (i=0 ; i<4 ; i++)
        if (levee[i].cb_couleur == demande)
            if (max == NONE)
                max = i;
            else if (plus_fort(levee[i].cb_valeur, levee[max].cb_valeur))
                max = i;

    if (max == NONE)
        fatal("levee invalide");

    return levee[max];
}
```

on y rencontre des accès à des champs d'un tableau de structures...

•)

Question 1.4 Proposez la définition d'un type de données pour mémoriser les cartes gagnées par un joueur lors des différentes levées de la partie.

(• ***Éléments de solution 1.4***

– C'est un ensemble d'au plus 32 cartes (un tableau) dont on mémorise aussi le cardinal. Deux informations à conserver donc une structure :

```
/* les levees d'un joueur */
struct levees_s {
    struct carte_belote_s le_cartes[32];
    int le_ncartes;
};
```

Un classique du C, des tableaux de taille variable (mais bornée).

•)

Exercice 2 (Grands entiers positifs, inspiré d'un énoncé de Christian QUEINNEC, LIP6**)**
On propose de manipuler de grands entiers positifs. Un grand entier positif est représenté par un vecteur de `GEP_SIZE` chiffres qui sont des `unsigned long`.

Question 2.1 Proposez la définition d'un type de données pour mémoriser un grand entier positif.

(• ***Éléments de solution 2.1*** Une structure (on ne connaît que ce moyen de définir un type...) contenant un seul tableau :

```
#define GEP_SIZE    3

struct gep_s {
    unsigned long gep_digits[GEP_SIZE];
};
```

•)

Question 2.2 Donnez le prototype d'une fonction qui retourne la somme de deux GEP et l'éventuelle retenue.

(• ***Éléments de solution 2.2***

- On ne peut retourner deux résultats (pour le moment, on a pas vu le passage d'adresses...).
- On crée donc un type associant un GEP et une retenue :

```
struct gep_carry_s {
    struct gep_s gep;
    int carry;                  /* 0 ou 1 */
};
```

- La fonction retourne donc une valeur de ce type :

```
struct gep_carry_s gep_add(struct gep_s a, struct gep_s b);
```

•)

Question 2.3 Proposez une définition de cette fonction.

(• ***Éléments de solution 2.3***

- La difficulté est de gérer la retenue. En additionnant deux valeurs entières positives x et y, et une retenue antérieure r (de valeur 0 ou 1), sans se préoccuper de dépassement,
  - si la somme $x + y + r$ est strictement supérieure à x, il n'y a pas de nouvelle retenue ;
  - si la somme $x + y + r$ est supérieure ou égale à x et que la valeur de la retenue antérieure r était 1, il n'y a pas de nouvelle retenue ;
  - sinon, il y a retenue.

  On itère cela pour chacun des chiffres des GEP :

```
struct gep_carry_s
gep_add(struct gep_s a, struct gep_s b)
{
    struct gep_carry_s c;
    int i;

    c.carry = 0;

    for (i=0; i<GEP_SIZE; i++) {
        c.gep.digits[i] = a.digits[i] + b.digits[i] + c.carry;
        c.carry =    (c.gep.digits[i] > a.digits[i])
                  || (    (c.gep.digits[i] == a.digits[i])
                       && (c.carry ==0)) ? 0 : 1;
    }

    return c;
}
```

  on peut légitimement préférer employer un `if` :

```
struct gep_carry_s
gep_add_2(struct gep_s a, struct gep_s b)
{
    struct gep_carry_s c;
    int i;

    c.carry = 0;

    for (i=0; i<GEP_SIZE; i++) {
        c.gep.digits[i] = a.digits[i] + b.digits[i] + c.carry;
        if (   (c.gep.digits[i] > a.digits[i])
            || (    (c.gep.digits[i] == a.digits[i])
                 && (c.carry ==0)))
            c.carry = 0;
```

```
            else
                c.carry = 1;
        }

        return c;
    }
```

•)

Exercice 3 (Manipulation de formules)
On désire manipuler des formules caractérisées par :

- le résultat d'une formule est une valeur flottante ;
- une formule peut être un terme ;
- une formule peut être la somme de deux termes ;
- une formule peut être le produit (ou le quotient) d'un terme et d'un coefficient entier ;
- un terme est soit une valeur flottante immédiate, soit un nom de variable ;
- une variable est un identificateur auquel est associée une valeur flottante.

Ainsi les formules suivantes sont valides :

$$5.0 \qquad a \qquad a + 4.0 \qquad a + b \qquad b \times 3$$

On propose les définitions et déclarations suivantes pour les variables et les termes :

```
/* Les variables */
#define NAME_LG    16           /* longueur d'un id de variable */
#define NVARS      32           /* nombre de variables */

struct var_s {
    char  v_name[NAME_LG];      /* nom de la variable */
    float v_val;                /* valeur de cette variable */
};

static struct var_s vars[NVARS]; /* "table des symboles" */
static unsigned int nvars=0;

/* Les termes */
enum term_type_e {VAL_T, ID_T}; /* deux types de termes */

struct term_s {
    enum term_type_e        t_type;
    union {
        float t_val;
        char  t_var[NAME_LG];
    }                       t_term;
};
```

Question 3.1 Proposez une fonction `value_of_term()` qui retourne la valeur d'un terme.
(• ***Éléments de solution 3.1***

- Rien à dire sur la déclaration du tableau de variables, si ce n'est que le mot clé `static` apparaît ici : limitation de la portée au « module ».
- On peut écrire la définition d'un terme en n'usant pas de définition d'union anonyme :

```
/* Les termes sans union anonyme */
union term_v_u {
    float t_val;
    char  t_var[NAME_LG];
};
```

```
struct term_s {
    enum term_type_e t_type;    /* discriminant */
    union term_v_u   t_term;    /* partie variable */
};
```

c'est peut être plus facile pour une première approche, mais n'apporte rien que des soucis de trouver un nom qui ne sera jamais (`term_v_u` utilisé car cette partie variante n'apparaîtra pas dehors de `term_s`.

– La fonction `value_of_term()` est déclarée :

```
float value_of_term(struct term_s t);
```

– On se définit la fonction intermédiaire

```
float
value_of_var(char *var)
{
    int i;

    for (i=0; i<nvars; i++) {
        if (!strcmp(vars[i].v_name, var))
            return vars[i].v_val;
    }

    return (float) fatal("variable inconnue");
}
```

– En passant j'utilise une version basique de mon `fatal()` :

```
/* will not return but exit.
   return an int in order you can return fatal() in a non void function. */
extern int fatal(const char *fmt, ...);
```

– L'implantation de `value_of_term()` illustre l'utilisation typique d'une union :

```
float
value_of_term(struct term_s t)
{
    switch (t.t_type) {
        case ID_T :
            return t.t_term.t_val;
        case VAL_T :
            return value_of_var(t.t_term.t_var);
    }

    /* makes gcc -Wall happy */
    return (float) fatal("dans du code inaccessible");
}
```

•)

Question 3.2 Proposez une définition de type pour représenter une formule.

(• ***Éléments de solution 3.2***

– On propose d'emblée une solution avec union anonyme :

```
enum op_e {NONE_OP, SUM_OP, MUL_OP, DIV_OP};

struct form_s {
    struct term_s            f_operand1;
    enum op_e                f_operator;
    union {
        struct term_s f_term;
        int           f_coeff;
    }                        f_operand2;
};
```

•)

Question 3.3 Proposez une fonction `value_of_formula()` qui retourne la valeur de l'évaluation d'une formule passée en paramètre.

(• ***Éléments de solution 3.3***

– Dans la même veine que `value_of_term()` :

```
float
value_of_formula(struct form_s f)
{
    switch (f.f_operator) {
        case NONE_OP :
            return value_of_term(f.f_operand1);
        case SUM_OP :
            return value_of_term(f.f_operand1)
                + value_of_term(f.f_operand2.f_term);
        case MUL_OP :
            return value_of_term(f.f_operand1)
                * f.f_operand2.f_coeff;
        case DIV_OP :
            return value_of_term(f.f_operand1)
                / f.f_operand2.f_coeff;
    }

    /* makes gcc -Wall happy */
    return (float) fatal("dans du code inaccessible");
}
```

•)