Programmation en C
Les structures

Régis Barbanchon

L1 Info-Math, Semestre 2

Définir un type structuré – Syntaxe Générale

La définition d'un type structuré se fait avec le mot-clé `struct`.

La syntaxe est la suivante :

```
struct TypeTag {
  Type1 field_1;
  ...
  TypeN field_n;
} var_1, ..., var_m;
```

- Elle déclare qu'il existe un type structuré `struct TypeTag`.
- elle définit ce type comme un agrégat de n champs `<field_1,...,field_n>`, où `field_k` est de type `TypeK`.
- elle déclare m variables `var_1,...,var_m` de ce type.

Définir d'un type structuré – Exemple : les points

Représentons les points du plan par leurs coordonnées cartésiennes.
Cela peut se faire par la définition du type structuré suivant :

```
struct Point {
  double x;
  double y;
} p, q;
```

- on a déclaré qu'il existe un type structuré `struct Point`.
- on a défini ce type comme un doublet `<x,y>` de flottants.
- on a déclaré 2 variables `p, q` de type `struct Point`.

On peut aussi regrouper les champs de même type, et écrire :

```
struct Point {
  double x, y;
} p, q;
```

Ne jamais déclarer de variables lors de la définition du type

Le plus souvent :

- les types sont définis en dehors de toute fonction.
- on ne déclare jamais de variables lors de la définition de type (elle seraient alors globales).

On écrit donc toujours :

```
struct TypeTag {
  Type1 field_1;
  ...
  TypeN field_n;
};
```

Par exemple, dans le cas des points du plan, on écrit :

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

Attention : le point-virgule derrière l'accolade doit rester présent.

Autres exemples de définitions de types structurés

Une date formée par trois entiers (jour, mois, année) :

```
struct Date {
  int day, month, year;
};
```

Un nom de personne complet formé de deux chaînes :

```
#define NAME_SIZE 128
struct FullName {
  char first [NAME_SIZE], last [NAME_SIZE];
};
```

Un tirage partiel de loterie, formé du tableau des n° déjà sorties :

```
#define LOTTO_SIZE 6
struct Lotto {
  int balls [LOTTO_SIZE], ball_count;
};
```

Un cercle défini par son centre (aussi une structure) et son rayon :

```
struct Circle {
  struct Point center;
  double radius;
};
```

Déclarer des variables de type structuré

Une fois un type structuré défini, les variables de ce type se déclarent avec la syntaxe :

```
struct TypeTag var_1, ..., var_m;
```

Par exemple, avec les types dèjà définis...

```
int main (void)
{
  struct Point p, corners[3];
  struct FullName name;
  struct Lotto lotto;
  struct Circle c;
  ...
}
```

...4 variables structurées sont déclarées dans la fonction `main()`, ainsi qu'un tableau `corners[]` de 3 structures.

Accéder aux champs d'une variable structurée

On accède au champ `field` d'une variable `var` par la syntaxe :

```
var.field
```

. . . que ce soit pour écrire la valeur du champ ou la lire :

```
int main (void) {
  struct Point p;
  p.x= 5.0; p.y= 8.0;                    // set fields
  printf ("p is <%f, %f>\n", p.x, p.y); // get fields

  struct FullName name;
  strcpy (name.first, "Regis");
  strcpy (name.last, "Barbanchon");
  printf ("name is <%s %s>\n", name.first, name.last);
  return 0;
}
```

La sortie produite par cet exemple est :

```
p is <5.0, 8.0>
name is <Regis Barbanchon>
```

On peut chaîner les opérateurs “.” et “[]”

Ils ont même priorité et associent de gauche à droite, et donc...

- on peut faire suivre un point “.” par un “.” :

```
struct Circle c;
c.center.x= 5.0;          // (c.center).x= 5.0;
c.center.y= 8.0;          // (c.center).y= 8.0;
```

- on peut faire suivre un crochet “[ ]” par un point “.” :

```
struct Point corners[3];
corners[0].x= 5.0;        // (corners[0]).x= 5.0;
corners[0].y= 8.0;        // (corners[0]).y= 8.0;
```

- on peut faire suivre un point “.” par un crochet “[ ]” :

```
struct Lotto lotto;
lotto.ball_count= 2;
lotto.balls[0]= 34;       // (lotto.balls)[0]= 34;
lotto.balls[1]= 26;       // (lotto.balls)[1]= 26;
```

Utiliser = pour recopier une structure dans une autre

L'opérateur d'affectation = fonctionne entre structures, dès lors que la source et la destination ont le même type :

```
int
main (void)
{
  struct Point p, q;
  p.x= 5.0; p.y= 8.0;

  q= p; // structure assigned by another structure
  printf ("q is <%f,%f>\n", q.x, q.y);
  return 0;
}
```

La valeur des champs de la source sont recopiés dans les champs correspondants de la destination :

```
q is <5.0, 8.0>
```

Il n'y a pas d'opérateur == ou != pour les structures

Étant données deux structures dont on veut tester l'égalité :

```
struct Date d1, d2;
d1.day= 25; d1.month= 12; d1.year= 2015;
d2.day= 25; d2.month= 12; d2.year= 1984;
```

On ne peut malheureusement pas écrire simplement :

```
if (d1 == d2) printf("d1 == d2\n"); // syntax error
```

```
if (d1 != d2) printf("d1 != d2\n"); // syntax error
```

Il faut donc tester l'égalité ou l'inégalité champ à champ :

```
if (d1.day   == d2.day   &&
    d1.month == d2.month &&
    d1.year  == d2.year) printf("d1 == d2\n");
```

```
if (d1.day   != d2.day   ||
    d1.month != d2.month ||
    d1.year  != d2.year) printf ("d1 != d2\n");
```

(1/2) Le test d'égalité n'est de toute façon pas toujours ==

Voici deux fonctions `f()` et `g()` mathématiquement réciproques :

```
double f (double x) { return 1.8 * x + 32.0;   } // Celcius to Fahrenheit
double g (double x) { return (x - 32.0) / 1.8; } // Fahrenheit to Celcius
```

Idéalement, on devrait avoir les trois égalités :
`x == f(g(x))`, `f(g(x)) == g(f(x))`, `g(f((x)) == x`.

```
int main (void) {
  double x= 2.0;
  printf ("%.18f\n", x);
  printf ("%.18f\n", f(g(x)));
  printf ("%.18f\n", g(f(x)));
  return 0;
}
```

Mais les approximations de calcul font que ce n'est pas le cas :

```
2.000000000000000000
1.999999999999997113
2.000000000000000888
```

(2/2) Le test d'égalité n'est de toute façon pas toujours ==

On teste donc généralement l'égalité de deux flottants à ε-près...

```
#define EPSILON 0.0000001

bool Double_IsEqual (double self, double other) {
  return fabs (self - other) < EPSILON;
}
```

Et cela se répercute par exemple sur le test d'égalité des points :

```
struct Point p;   p.x= 5.0;          p.y= 8.0;
struct Point q;   q.x= f(g(p.x));    q.y= f(g(p.y));
```

```
if (  Double_IsEqual (p.x, q.x) &&
      Double_IsEqual (p.y, q.y)) {
  printf ("p == q\n");
}
```

```
if (! Double_IsEqual (p.x, q.x) ||
    ! Double_IsEqual (p.y, q.y)) {
  printf ("p != q\n");
}
```

Initialiser une variable structurée en C89

On initialise tous les champs d'une variable `var` avec la syntaxe :

```
struct TypeTag var= { value_1, ..., value_n };
```

Exemple avec un point du plan :

```
struct Point p= { 5.0, 8.0 };
```

- possible uniquement à la déclaration de la variable :

```
struct Point p;
p= { 5.0, 8.0 }; // illegal
```

- les valeurs `value_k` doivent être des constantes littérales :

```
double some_x= 5.0, some_y= 8.0;
struct Point p= { some_x, some_y }; // illegal
```

- fragile : dépendant de l'ordre de définiton des champs.

Initialiser une variable structurée en C99

Les *"compound literals"* y ont été introduits avec la syntaxe :

```
(struct TypeTag) {
  .field_1= expr_1, ...,
  .field_n= expr_n,
}
```

- possible en dehors de toute déclaration :

```
struct Point p;
p= (struct Point) { .x= 5.0, .y= 8.0, };
```

- les expressions `expr_k` sont arbitraires :

```
double a= M_PI/3.0;
p= (struct Point) { .x= cos(a), .y= sin(a), };
```

- ne dépend pas de l'ordre de définition des champs :

```
p= (struct Point) { .y= 8.0, .x= 5.0, };
```

(1/2) Retourner une structure depuis fonction

On retourne une structure comme on retourne un type primitif :

```
struct Point
Point_Make (int x, int y) {
  Point self;
  self.x= x;
  self.y= y;
  return self;
}
```

On peut aussi retourner directement un *“compound literal”* :

```
struct Date
Date_Make (int day, int month, int year) {
  return (struct Date) {
    .day= day,
    .month= month,
    .year= year,
  };
}
```

(2/2) Récupérer la structure retournée par une fonction

L'opérateur d'assignement "=" fonctionnant sur les structures, on récupère le résultat de la même manière qu'un type primitif :

```
int main (void)
{
  struct Point origin= Point_Make (0.0, 0.0);
  struct Date    xmas=  Date_Make (25, 12, 2015);
  printf ("origin: <%f, %f>\n",
          origin.x, origin.y);
  printf ("xmas: %d/%d/%d\n",
          xmas.day, xmas.month, xmas.year);
  return 0;
}
```

La variable locale `self` retournée est recopiée dans `origin`, et le *"compound literal"* retourné est recopié dans `xmas` :

```
origin: <0.0, 0.0>
xmas: 25/12/2015
```

(1/3) Paramétrer une fonction avec des structures

On peut paramétrer une fonction avec des structures,
de la même manière qu'on le fait avec les types primitifs :

Pour tester que deux dates sont égales :

```
bool
Date_IsEqual (struct Date self, struct Date other)
{
  return self.day   == other.day
    &&   self.month == other.month
    &&   self.year  == other.year;
 }
```

Pour convertir une date en chaîne de caractères :

```
void
Date_FillString (struct Date self, char string[])
{
   sprintf (string, "%d/%d/%d\n",
            self.day, self.month, self.year);
}
```

(2/3) Passer des structures en argument à une fonction

Les arguments sont copiés dans les paramètres formels des fonctions, de la même manière que le sont les types primitifs :

```
int main (void) {
  struct Date xmas= Date_Make (25, 12, 2015);
  struct Date today= xmas;

  char today_string [50];
  Date_FillString (today, today_string);
  printf ("today is %s\n", today_string);

  if (Date_IsEqual (today, xmas))
    printf ("Merry X-Mas!\n")
  return 0;
}
```

Ainsi, l'argument `today` est copié dans le paramètre formel `self`, et l'argument `xmas` est copié dans le paramètre formel `other` :

```
today is 25/12/2015
Merry X-mas!
```

(3/3) Et si une fonction modifie un paramètre formel ?

Comme tous les types, les structures sont passées par copie :

```
struct Date
Date_FirstDayOfNextMonth (struct Date self) {
  self.day= 1; self.month ++;
  if (self.month <= 12) return self;
  self.month= 1; self.year ++;
  return self;
}
```

```
int main (void) {
  struct Date xmas= Date_Make (25, 12, 2015);
  struct Date next= Date_FirstDayOfNextMonth (xmas);
  char xmas_str[50]; Date_FillString (xmas, xmas_str);
  char next_str[50]; Date_FillString (next, next_str);
  printf ("xmas: %s, next: %s\n", xmas_str, next_str);
  return 0;
}
```

`xmas` n'est pas modifié car le paramètre `self` en est une copie :

```
xmas: 25/12/2015, next: 1/1/2016
```

(1/3) Se débarrasser du mot-clé `struct` avec `typedef`

La directive `typedef` pemet de créer un synonyme/alias de type :

```
typedef   ExistingType   SynonymType;
```

Par exemple, pour que `ULong` soit synonyme de `unsigned long` :

```
typedef   unsigned long   ULong;
```

```
int main (void) {
  ULong number= 0x1000000UL;
  printf ("%lu\n", number);
  return 0;
}
```

Et pour que `Date` soit synonyme de `struct Date` :

```
typedef   struct Date   Date;
```

```
int main (void) {
  Date xmas= Date_Make (25, 12, 2015);
  ...
}
```

(2/3) Se débarrasser du mot-clé `struct` avec `typedef`

On écrit rarement la définition du type et son alias en 2 temps :

```
struct Date {
  int day, month, year;
};
```

```
typedef   struct Date   Date;
```

On procède plutôt en 1 seul temps avec la syntaxe combinée :

```
typedef struct Date {
  int day, month, year;
} Date;
```

Et si on n'utilise plus le tag du type, on peut même écrire :

```
typedef struct {
  int day, month, year;
} Date;
```

(3/3) Se débarrasser du mot-clé `struct` avec `typedef`

Dans `Date_Make()`, l'alias simplifie le typage du retour :

```
struct Date
Date_Make (int day,
           int month,
           int year)
{
  return (struct Date) {
    .day= day,
    .month= month,
    .year= year,
  };
}
```

⇒

```
Date
Date_Make (int day,
           int month,
           int year)
{
  return (Date) {
    .day= day,
    .month= month,
    .year= year,
  };
}
```

Dans `Date_IsEqual()`, l'alias simplifie le typage des paramètres :

```
bool
Date_IsEqual (struct Date self,
              struct Date other)
{
  return self.day   == other.day
    &&   self.month == other.month
    &&   self.year  == other.year;
 }
```

⇒

```
bool
Date_IsEqual (Date self,
              Date other)
{
  return self.day   == other.day
    &&   self.month == other.month
    &&   self.year  == other.year;
 }
```