12. Structures

On présente ici quelques éléments de base pour définir des structures simples en C. Pour des cas plus complexes il vaut mieux utiliser les classes du C++.

1 Définition

Une structure est un ensemble d'éléments de types variés tels que constantes, variables, pointeurs, regroupés sous un même nom. Ce nom, choisi par l'utilisateur, permet, au choix, de traiter l'ensemble comme un seul bloc, ou chaque élément séparément. Pour les structures les éléments sont appelés « membres » ou « champs » . Ici on utilisera le terme « membres » .

2 Déclaration

2.1 Déclaration de base, membres

On déclare le type de la structure de la façon suivante, par exemple :

```
struct stru {
  int i;
  double x, y;
  char c;
};
```

`struct` est un mot clé du C, `stru` le nom choisi par l'utilisateur pour le type de la structure, `i`, `x`, `y` et `c` les noms des membres choisis par l'utilisateur. Dans cet exemple la structure comprend un `int`, deux `double` et un `char`, on pourrait évidemment en mettre plus, ou ajouter des pointeurs de tous types.
`i`, `x`, `y` et `c` ne sont pas des variables, ce sont des noms de membres.
On déclare ensuite une structure du type `stru` par :

```
struct stru ss;
```

`ss` étant le nom de la structure choisi par l'utilisateur. On peut alors désigner la structure dans son ensemble par son nom, ou ses membres séparément avec l'opérateur « . » : `ss.i` est la variable de type `int` contenue dans la structure `ss`, `ss.x` est la première variable de type `double` contenue dans la structure `ss`, etc..

2.2 Déclaration simplifiée avec `typedef`

Si on ajoute à la déclaration de structure précédente :

```
typedef struct stru struty;
```

alors toutes les déclarations telles que :

```
struct stru ss;
```

peuvent être simplifiées en :

```
struty ss;
```

Comme d'habitude, le nom du type (ici `struty`) peut être choisi librement par l'utilisateur. Par contre, `typedef` est un mot clé du C.
Comme pour les variables ordinaires on peut déclarer plusieurs structures du même type simultanément :

```
struty ss1, ss2, ss3;
```

On peut combiner la déclaration du type de la structure et le `typedef` en une seule commande (ce qui est en fait la façon préférée de déclarer un type de structure) :

```
typedef struct {
  int i;
  double x, y;
  char c;
} struty;
```

On voit que dans ce cas il n'est pas nécessaire d'introduire le nom `stru` intermédiaire.

2.3 Règles

Un membre d'un type structure peut être d'un type structure défini par ailleurs, mais pas du type structure auquel il appartient.
Si un type de structure est déclaré dans une fonction (`main` compris), ce type n'est accessible que dans cette fonction. S'il est déclaré en dehors de toute fonction il est accessible depuis toute fonction située après cette déclaration (même règle que pour l'accessibilité d'une variable).
Contrairement au cas des variables il n'est pas possible d'accéder à un type de structure déclaré dans un fichier différent.

3 Utilisation

3.1 Initialisation

Exemple d'initialisation :

```
#include<iostream>
using namespace std;
int main() {
  typedef struct {
    int i;
    double x, y;
    char c;
  } struty;
  struty ss = {5, 1.26, 2.34, 'K'};
  cout << ss.i << " " << ss.x << " " << ss.y << " " << ss.c << endl;
  return 0;
}
```

3.2 Affectation : utilisation du signe =

On peut affecter la valeur d'une structure à une autre structure avec le signe = comme pour des variables simples en écrivant :

```
ss2 = ss1;
```

à condition, bien entendu, que `ss1` ait été initialisée auparavant et que les deux structures `ss1` et `ss2` soient du même type (ici `struty`). La valeur de chaque membre de `ss1` est alors attribuée au membre correspondant de `ss2`. Les membres étant des variables comme les autres rien n'interdit d'écrire également :

```
ss2.i = ss1.i;
ss2.x = ss1.x;
...
```

pour tout ou partie des membres de `ss1` et `ss2`.
On ne peut, par contre, pas comparer deux structures de même type à l'aide des opérateurs == ou !=, il faut comparer membre à membre.

3.3 Adresse et taille d'une structure

L'adresse s'obtient comme pour une variable simple, en utilisant l'opérateur &.
L'opérateur `sizeof` donne la taille de la structure, qui est égale ou supérieure à la somme de celles de ses membres.

3.4 Passage d'une structure en argument d'une fonction

Comme pour une variable simple on peut passer par valeur ou par adresse.

3.4.1 Passage par valeur

```
... f(..., struty zz, ...) {
  ...
}
int main() {
  ...
  struty ss;
  ...
  f(..., ss, ...);
  ...
}
```

Même si les membres de `zz` sont modifiés ceux de `ss` ne le sont pas.

3.4.2 Passage par adresse

```
... f(..., struty *zz, ...) {
  ...
}
int main() {
  ...
  struty ss;
  ...
  f(..., &ss, ...);
  ...
}
```

Dans la fonction `f`, `zz` n'est plus une structure mais un pointeur sur une structure. Pour accéder à la structure dont l'adresse se trouve dans le pointeur (c'est à dire ici à la structure `ss`), il faut écrire `*zz` et pour accéder aux membres de cette structure `(*zz).i` [1], `(*zz).x`, etc.. Il existe une autre notation pour désigner `(*zz).i` qui est `zz->i`, et qui permet donc d'accéder aux membres d'une structure en utilisant son adresse de début.
Si les membres de `*zz` sont modifiés ceux de `ss` le sont.

3.5 Fonction de type structure

Une fonction peut être de type structure :

```
struty f(...) {
  struty xx;
  ...
  return xx;
}
```

4 Application : dioptre sphérique

On considère un dioptre sphérique dont l'axe optique est orienté dans le sens de propagation de la lumière.

1. Les parenthèses sont indispensables car « . » est prioritaire sur « * » .

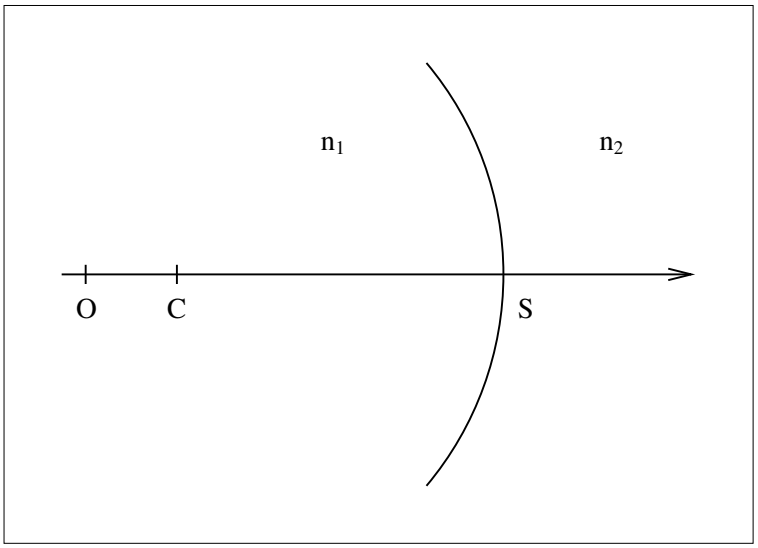

FIGURE 1 –

x est l'abscisse du sommet S du dioptre par rapport à une origine O quelconque de l'axe, et C étant le centre de courbure du dioptre, on pose $r = \overline{SC}$ (rayon de courbure algébrique). On note n_1 l'indice avant le dioptre, n_2 après. Le programme suivant permet de définir un dioptre par ses quatre paramètres x, r, n_1, n_2 et calcule sa matrice de transfert. On pourrait le développer en définissant un ensemble de dioptres dans un tableau de dioptres et en calculant la matrice de transfert du système total, puis faire tracer le trajet des rayons lumineux à travers les dioptres successifs, étudier l'aberration chromatique.

```
// Ce programme calcule la matrice d'un dioptre
#include<iostream>
#include<stdlib.h>
using namespace std;
//-----------------------------------------------------------------------
typedef struct {
  double x, sc, n1, n2;
} diop;
//-----------------------------------------------------------------------
void matrice(diop a, double **mm) {
  mm[0][0] = 1.;
  mm[0][1] = 0.;
  mm[1][0] = (a.n1 - a.n2) / a.n2 / a.sc;
  mm[1][1] = a.n1 / a.n2;
}
//-----------------------------------------------------------------------
int main() {
  const int nn = 2;
  int i, j;
  diop d = {1., -1., 1., 1.5};
  double** mm = (double**)malloc(nn * sizeof(double*));
  for(i = 0; i < nn; i++)
    mm[i] = (double*)malloc(nn * sizeof(double));
  // ou double** mm = D_2(nn,nn); en utilisant les fonctions du Magistère
  matrice(d,mm);
  for(i = 0; i < nn; i++) {
    for(j = 0; j < nn; j++)
      cout << mm[i][j] << " ";
    cout << endl;
  }
```

```
  return 0;
}
```

5 Tableau dynamique dans une structure

Exemple de tableau dynamique dans une structure :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  typedef struct {
    double* x;
  } struty;
  int i, n = 5;
  struty z;
  z.x = (double*)malloc(n * sizeof(double));
  for(i = 0; i < n; i++)
    z.x[i] = i + 1.5;
  for(i = 0; i < n; i++)
    cout << z.x[i] << " ";
  cout << endl;
  return 0;
}
```

6 Tableau dynamique de structures

Exemple de déclaration d'un tableau dynamique de structures :

```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main() {
  typedef struct {
    int i;
    double x, y;
    char c;
  } struty;
  int n = 5;
  struty* w = (struty*)malloc(n * sizeof(struty));
  // On a alors n structures du type struty : w[0] ... w[k] ... w[n-1] avec k=0 ... n-1
  // Les membres s'obtiennent par : w[k].i, w[k].x, w[k].y, w[k].c
  return 0;
}
```