[Math-L312] TP 8 : Structures et algorithmes

Adrien Semin
adrien.semin@math.u-psud.fr

1 Introduction : pourquoi utiliser des structures ?

Imaginons qu'on veuille implémenter le type complexe en `C`. Une première idée est d'utiliser deux variable de type `double`, mais celà demande beaucoup d'attention de la part du programmeur. Une deuxième idée est d'utiliser un tableau fixe à deux entrées `double[2]`, mais cel'a demande également de l'attention de la part du programmeur, et cette astuce ne fonctionne pas si on veut par exemple représenter des personnes. Une troisième idée est d'utiliser une structure : une variable qui contiendrait à la fois la partie réelle du complexe et la partie imaginaire du complexe. Typiquement, la déclaration d'une structure se fait par :

```
typedef struct {
  type1 nom1;
  type2 nom2;
  ...
  typeN nomN;
} nom_structure;
```

Exemple pour les nombres complexes :

```
typedef struct {
  double re;
  double im;
} complexe;
```

A partir de là, nous pouvons faire sur la structure que nous avons déclaré les opérations standard que nous pourrions faire sur un type classique :

- affectation :

```
complexe i = {0,1};
complexe a = i;
```

- utilisation dans des fonctions

```
double abs(complexe c)
{
  double v_abs = sqrt(c.re*c.re+c.im*c.im);
  return v_abs;
}
```

  On remarquera que l'accès au membre `re` du type `complexe` se fait par l'opérateur unaire '.'

- pointeur

```
complexe* Cplx;
// ...
printf("\%f",(*Cplx).re;
printf("\%f",Cplx->re;
// Les deux instructions ci-dessus sont \'equivalentes
```

`* Question orale *` Quelle est la différence entre les deux éléments suivants

```
(*a).b
*a.b
```

– création de tableaux : on pourra regarder le programme suivant, téléchargeable à l'adresse `/home/doc/semin/TP/c/tableau_c.c` :

```c
#include<time.h>
#include<stdio.h>
#include<stdlib.h>
#include<allocation_tableaux.h>
#include<assert.h>


typedef struct {
  double re;
  double im;
} complexe;

void remplissage(complexe*,int);

void print(complexe*,int);

int main(int argc, char* argv[])
{
  if (argc != 2) // Si a l'execution, on n'a pas passe un parametre
    {
      printf("Usage : %s nombre\n",argv[0]);
      exit(1);
    }
  // Initialisation du generateur aleatoire
  srand(time(NULL));
  // Recuperation de la taille du tableau
  int N; assert(sscanf(argv[1],"%d",&N)==1);
  // Declaration du tableau
  allocation_tableau(Cplx , complexe , N);
  remplissage(Cplx, N);
  print(Cplx, N);
  return 0;
}

void remplissage(complexe* c, int N)
{
  int i;
  for(i=0 ; i<N ; i++)
    {
      c[i].re = rand() % 10 + 1;
      c[i].im = rand() % 10 + 1;
    }
}

void print(complexe* c, int N)
{
  int i;
  for(i=0 ; i<N ; i++)
    {
      printf("c[%d]=(%f)+(%f).i\n",i,c[i].re,c[i].im);
    }
}
```

2 Travail à rendre

1. `calcul_complexe.c` ***À Rendre!*** A partir de la structure de nombre complexe, implémenter les opérations suivantes dans le programme `calcul_complexe.c` :

– l'addition de deux nombres complexes,
– la soustraction de deux nombres complexes,
– la multiplication d'un nombre complexe par un nombre réel,
– la multiplication de deux nombres complexes,
– la division de deux nombres complexes.

Les prototypages associés seront successivement

```
complexe add_c(complexe, complexe);
complexe sub_c(complexe, complexe);
complexe mult_cr(complexe, double);
complexe mult_c(complexe, complexe);
complexe div_c(complexe, complexe);
```

Tester également les opérations avec $a = 3 + 2\imath$ et $b = -1 + 4\imath$

2. `tri_tableau.c` ∗ ***À Rendre!*** ∗ Dans le programme `tri_tableau.c`, implémenter les fonctions de tri suivantes (on souhaite trier un tableau de `long` par ordre croissant) :
   – fonction de tri direct : la fonction de tri direct se fait de la manière suivante :
     – on recherche l'élément le plus grand du tableau, et on met cet élément à la dernière place,
     – on recherche le second plus grand élément du tableau, et on met cet élément à l'avant-dernière place,
     – on recherche le troisième plus grand élément du tableau, et on met cet élément à l'avant-avant-dernière place,
     – etc.
   – fonction de tri à bulle : le tri à bulle se programme de la manière algorithmique suivante (N désignant la taille du tableau que l'on veut trier) :
     (i) on parcourt pour i entre 1 et $N-1$,
     (ii) si l'élément i du tableau est plus grand que l'élément $i+1$, on inverse ces deux éléments
     (iii) on reprend (i) si au moins une inversion a été faite dans (ii).
     ∗ ***Question orale*** ∗ Pourquoi cet algorithme peut être considéré comme meilleur que l'algorithme de tri direct.
   – fonction de tri-fusion : on trie séparément les deux demi-tableaux, et on fusionne les deux demi-tableaux de manière à ce que le tableau final soit bien trié par ordre croissant. Bien entendu, on triera chaque sous-tableau par un algorithme de tri-fusion, à moins que le sous-tableau soit de taille 1, auquel cas il est déjà trié.
     ∗ ***Question orale*** ∗ Si C_N exprime la complexité pour trier un tableau de taille N, comment s'exprime C_N en fonction de $C_{N/2}$?

   Les prototypes des fonctions seront définies par

```
void tri_direct(long*,int);
void tri_bulle(long*,int);
void tri_fusion(long*,int);
```

   ∗ ***Question orale*** ∗ Pourquoi passe-t-on également un argument de type `int` pour chacune des trois fonctions ?

   On testera ce programme sur un tableau aléatoire de 2^p nombres de type `long`, où p sera un entier passé en paramètre lors de l'exécution.

3. lissage.c Ecrire un programme `lissage.c` qui prendra un nom d'image `pgm` d'entrée, un nom d'image `pgm` de sortie et un nombre réel $\alpha \in [0, \frac{1}{4}[$ à l'exécution et qui construira l'image de sortie de la manière suivante : si on note respectivement N_H, N_W le nombre de ligne et le nombre de colonnes de l'image d'entrée, pour $0 \leq i \leq N_H - 1$, pour $0 \leq j \leq N_W - 1$, si on note $e_{i,j}$ le pixel de la i-ème ligne et de la j-ème colonne de l'image d'entrée, alors on construira le pixel de la i-ème ligne et de la j-ème colonne de l'image de sortie $s_{i,j}$ par la relation

$$s_{i,j} = (1 - 4\alpha)e_{i,j} + \alpha(e_{i+1,j} + e_{i-1,j} + e_{i,j-1} + e_{i,j+1})$$

   Par convention, on considèrera que $e_{-1,\bullet} = e_{0,\bullet}$, $e_{N_H,\bullet} = e_{N_H-1,\bullet}$, $e_{\bullet,-1} = e_{\bullet,0}$ et $e_{\bullet,N_W} = e_{\bullet,N_W-1}$. Tester sur les images `damier.pgm`, `grey.pgm` et `lena.pgm` avec $\alpha = \frac{1}{16}$, $\alpha = \frac{1}{8}$ et $\alpha = \frac{3}{16}$. ∗ ***Question orale*** ∗ Que se passe-t-il quand l'image d'entrée est constante ?