**Université Pierre et Marie Curie** — **Master M1 "Mathématiques et Applications"**
**Période 2 - 2011/2012** — **Programmation en C/C++, MM056**

**TD 2 : Tableaux, structures et syntaxe**

Le but de ce TD est de trier un tableau de nombre complexe (du plus petit module au plus grand) en utilisant l'algorithme "Quicksort".

Rappel 1 : Pour déclarer le nom d'un type, on fait suivre le mot réservé `typedef` d'une expression identique à une déclaration de variable, dans laquelle le rôle du nom de la variable est joué par le nom du type qu'on veut définir. Exemple :

```
typedef double mes_matrices[3][2];
mes_matrices m = { {1,2}, {2,3}, {3,4}};
mes_matrices n = { 1, 2, 2, 3, 3, 4 };
```

Rappel 2 : `struct` définit des variables :

```
struct ma_structure { double real; double imag; int n;};
```

définit `ma_structure` comme un genre d'objet donnant lieu à deux réels et un entier. On l'utilise ensuite ainsi :

```
struct ma_structure { double real; double imag; int n;} z1, z2;
z1.real = 1; z1.imag = 3; z1.n=6; z2=z1;
```

ou bien

```
struct ma_structure { double real; double imag; int n;};
struct ma_structure z1, z2;
z1.real = 1; z1.imag = 3; z1.n=6; z2=z1;
```

ensuite, `typedef` fait de cette structure un type :

```
typedef struct { double real; double imag; int n;} le_type_de_ma_structure;
le_type_de_ma_structure z1, z2;
z1.real = 1; z1.imag = 3; z1.n=6; z2=z1;
```

intérêt de `typedef` ici : évite d'avoir à écrire `struct` à chaque fois.

Notons que une variable de type structure peut être initialisée au moment de sa déclaration :

```
struct ma_structure { double real; double imag; int n;} z={1, 3, 6};
```

ou bien

```
typedef struct { double real; double imag; int n;} le_type_de_ma_structure;
le_type_de_ma_structure z={1, 3, 6};
```

**1** *En utilisant le mot-clé* `typedef`*, définissez un nouveau type* `t_complex` *qui contient 2 champs*
- *1 champ nommé* `real` *de type* `double`
- *1 champ nommé* `imag` *de type* `double`

*qui contiennent respectivement la partie réelle et la partie imaginaire d'un complexe.*
*Écrire une fonction* `module` *qui calcule le module d'un nombre complexe.*
*Écrire un petit programme exécutable qui initialise une variable de type* `t_complex` *et qui affiche son contenu et son module.*

L'algorithme "Quicksort" peut s'écrire simplement à l'aide de 2 fonctions : une fonction récursive "quicksort" dont le pseudo-code est :

```
fonction quicksort(tableau, left, right)
    si left < right
        index_coupe := partition(tableau, left, right)
        quicksort(tableau, left, index_coupe-1)
        quicksort(tableau, index_coupe+1, right)
```

et une fonction “partition” définie comme ceci :

```
fonction partition(tableau, left, right)
    pivot := tableau[right]
    index_coupe := left
    pour i variant de left a right-1 faire
        si tableau[i] <= pivot
            echange tableau[i] et tableau[index_coupe]
            index_coupe := index_coupe + 1
    echange tableau[index_coupe] et tableau[right]
    renvoie index_coupe
```

**2** *Écrire une fonction* `partition_tab_comp` *et une fonction* `quicksort_tab_comp` *en s'inspirant de ces pseudo-codes. Attention, on compare les modules des nombres complexes pour les trier du plus petit au plus grand.*

Pour générer des nombres aléatoires vous pouvez utiliser la fonction `rand` définie dans la `stdlib`. Pour initialiser le générateur (une fois, au début du programme) il faut utiliser la fonction `srand`. Exemple :

```c
#include <stdlib.h>
#include <time.h>                       // necessaire pour la fonction time()

int main(void)
{
    double x;
    srand(time(NULL));                  // initialisation du generateur
    x = rand() / (RAND_MAX + 1.0);      // 0 <= x < 1
    printf("%g\n", x);
    return 0;
}
```

**3** *Construire un tableau de 20 nombres complexes dont la partie réelle et la partie imaginaire sont des nombres (pseudo-)aléatoires de* $[0, 1[$.
*Afficher le contenu du tableau, trier le tableau à l'aide de la fonction* `quicksort_tab_comp` *et afficher le tableau trié. (Afficher aussi le module, pour bien valider le tri.)*

Plus efficacement, on aurait pu construire un tableau composé des modules des nombres complexes et trier ce tableau.

**4** *Que pensez-vous de cette solution ? Que faut-il stocker en plus ? Créer de nouvelles fonctions* `quicksort_mod` *et* `partition_mod` *pour mettre en oeuvre cette approche.*
*Tester ces fonctions.*

Comme l'algorithme “Quicksort” a toujours la même structure quel que soit le type de tableau que l'on veut trier, il est plus efficace d'écrire une fonction générique (qui peut s'appliquer à tout type de tableau) en précisant la façon de comparer deux éléments du tableau.
Cette fonction générique `qsort` existe dans la `stdlib`. Voici une partie des informations sur la fonction `qsort` que l'on obtient en tapant `info qsort` :

```c
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

*La fonction* `qsort()` *trie une table contenant* `nmemb` *éléments de taille* `size`*. L'argument* `base` *pointe sur le début de la table.*
*Le contenu de la table est trié en ordre croissant, en utilisant la fonction de comparaison pointée par* `compar`*, laquelle est appelée avec deux arguments pointant sur les objets à comparer.*
*La fonction de comparaison doit renvoyer un entier inférieur, égal, ou supérieur à zéro si le premier argument est respectivement considéré comme inférieur, égal ou supérieur au second. Si la comparaison des deux arguments renvoie une égalité (valeur de retour nulle), l'ordre des deux éléments est indéfini.*

Par exemple, le programme ci-dessous utilise la fonction `qsort` pour trier un tableau d'entiers.

```c
#include <stdio.h>
#include <stdlib.h>

void affiche_tab_int(int tab_int[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%i\t", tab_int[i]);
    printf("\n");
}

int compare_int(void const *a, void const *b)
{
    int const *pa = a;
    int const *pb = b;
    return *pa - *pb;   // ou directement: return *(int*)a - *(int*)b;
}

int main (void)
{
    int tab[] = { 4, 6, -3, 4, 7 };
    affiche_tab_int(tab, 5);
    qsort(tab, 5, sizeof(int), compare_int);
    affiche_tab_int(tab, 5);
    return 0;
}
```

**5** *Reprendre les deux approches précédentes en utilisant la fonction* `qsort`.