TP3 : Fonctions génériques

Buts : donner un aperçu
- des fonctions génériques
- des Types de Données Abstraits

1 Installation

1. Récupérez l'archive `TP_Genericite.tar.gz` à partir :
   - de ma page *Enseignements* à `http ://www-info.univ-lemans.fr/~jacob/enseignement.html` dans la rubrique *Programmation C*
   - du serveur de l'IUP à `/info/tmp/AnnexesTPL2SPI/TP_Genericite/TP_Genericite.tar.gz`
2. Décompressez la et désarchivez la.
3. Comme dans le TP précédent, normalement tous les programmes et modules peuvent être compilés par `make -f Makefile all`

2 Cadre

Il faut concevoir dans ce TP un TDA Liste pour stocker des objets homogènes (du même type) mais dont ne connaît pas *a priori* le type. Pour tester cela les 3 mêmes objets que le TP précédent vous sont proposés
- `individu_t`
- `fraction_t`
- `string_t`

mais vous pouvez là aussi en créer d'autres.
Le but sera donc de faire des listes de ces objets (une par type donc) avec le même TDA Liste.
Comme dans le TP précédent , vous pouvez tester les fonctions de ces 3 types dans les programmes
- `test_individu`
- `test_fraction`
- `test_mystring`

3 Exercice 1 : Test de la liste

Le programme `test_liste` va créer des listes de `N` objets, `N` étant une constante (un `#define`). Modifiez `test_liste.c` pour qu'il accepte :

- un argument qui donne le nombre d'objets des listes
- une option `verbose` qui fera afficher tous les messages de trace si elle est positionnée. exemple de messages de trace

```
printf( "\nTest creation d'une liste de %d individus \n" , N ) ;
printf( "\nTest creation d'une liste de %d individus \n" , N ) ;
...
```

4 Exercice 2 : Affichage

Réaliser la fonction d'affichage de tous les éléments de cette liste. Le TDA liste ne sait pas *a priori* quelle fonction doit être utilisée pour afficher tous ses éléments. Pour répondre à ce problème, la fonction d'affichage (`liste_afficher`) aura en paramètre un pointeur sur la fonction d'affichage qu'il faudra utiliser pour tous ses éléments.

5 Exercice 3 : Destruction

Réalisez ensuite la fonction de destruction de tous les éléments de la liste, ainsi que la liste elle-même (on suppose que les éléments de la liste ne sont pas partagés). Le problème et sa solution sont les mêmes que pour celui de l'affichage des éléments : la fonction de destruction (`liste_detruire`) aura dans ses paramètres un pointeur sur la fonction qui est capable de le détruire (libérer sa place mémoire) de tous les éléments.

6 Exercice 4 : Tri

Réalisez enfin la fonction de tri d'une liste. L'ordre de tri ainsi que la comparaison des éléments seront fournis par une fonction externe : la fonction de tri (`liste_trier`) aura dans ses paramètres un pointeur sur la fonction qui est capable de comparer 2 éléments.
Vous utiliserez une des fonctions de tri de la bibliothèque `<stdlib.h>` pour trier votre liste, à savoir : `heapsort`, `mergesort` ou `qsort`.

7 Exercice 5 : Amélioration du tri

Modifiez la fonction `liste_trier` pour qu'elle accepte un nombre variable d'arguments : il s'agira d'un paramètre en surnombre qui indiquera quelle méthode choisir pour effectuer le tri. Si ce paramètre est du type ci-dessous :

```
typedef enum type_tri_s { QUICK , MERGE , HEAP } type_tri_t ;
```

alors dans la fonction

```
... type_tri_t type ; ...
switch( type )
{
   case QUICK: qsort(...) ;
   case MERGE: mergesort(...) ;
   case HEAP: heapsort(...) ;
}
```

Si cet argument n'est pas présent alors on effectue le `qsort`. On passera en argument du programme (`main`) la méthode choisie pour faire les tris des listes.