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 affichage de la liste \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. Le problème est le même que celui de l'affichage des éléments mais ici la fonction de destruction d'un élément sera dans les attributs de la liste et non dans les paramètres de `liste_detruire`. Il faudra donc ré-écrire la fonction de création d'une liste (`liste_creer`).

6 Exercice 4 : Ajout d'un élément

Ajoutez alors dans les attributs de la liste, une fonction qui affecte un élément dans une position de la liste. Ré-écrivez alors la fonction `liste_elem_ecrire` qui réalisait cette affectation jusque là par un `=`.

7 Exercice 5 : Tri bulle

Réalisez une fonction `liste_trier_bulle` qui trie une liste par la méthode du "tri bulle". 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 pouvez réutiliser l'exemple de la fonction de tri vue en cours.

8 Exercice 6 : Tri qsort

Réalisez une fonction `liste_trier_qsort` qui utilise la fonction pré-définie `qsort` pour faire le tri.

9 Exercice 7 : Amélioration du tri

Réalisez la fonction `liste_trier` ayant un nombre variable d'arguments : elle acceptera un seul argument 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 , PERSO } type_tri_t ;
```

alors dans la fonction `liste_trier`

```
... type_tri_t type ; ...
switch( type )
{
   case QUICK: appel de liste_trier_qsort(...) ;
   case PERSO: appel de liste_trier_bulle(...) ;
   ...
}
```

Si cet argument n'est pas présent alors on effectue le `qsort`. Le (`main`) aura alors un argument/option avec une valeur requise qui indiquera la méthode choisie pour faire les tris des listes. Par exemple :

```
-t0 ou --tri=0 pour la méthode qsort
-t1 ou --tri=1 pour la méthode bulle
```