S3E6 – Listes simplement chaînées *génériques*

LICENCE INFORMATIQUE : PROGRAMMATION NÉCESSAIRE

Stéfane

1 Introduction

Les listes dites *génériques* sont identiques aux listes d'entiers que nous avons vues jusqu'à maintenant.

Seule la donnée qui était un entier (`int`) devient une *référence à un objet dont le type n'est pas connu*: `void *`

On peut alors envisager deux types de listes:

- Les listes génériques *homogènes* dont les données sont toutes du même type,
- Les listes génériques *hétérogènes* dont les données sont de types différents.

Pour le projet, nous allons nous intéresser aux premières qui sont, de loin, les plus utilisées.

2 Listes génériques homogènes

Reprenez l'épisode S3:E3 sur les *listes simplement chaînées d'entiers*.

Modifiez ces listes d'entiers pour qu'elles deviennent des listes de *pointeurs sur quelque chose* : `void *`.

Le type abstrait des *éléments de liste* devient alors :

```
1  struct lst_elm_t {
2    void * datum;
3    struct lst_elm * suc;
4  }
5  /** @brief create a new list element and store datum into it */
6  struct lst_elm_t * new_lst_elm(void * datum);
7  /** @brief destroy the list element pointed by E and set E to NULL */
8  void del_lst_elm(struct lst_elm_t ** ptrE);
9  /** @brief retreive the datum of the list element E */
10 void * getDatum(struct lst_elm_t * E);
11 /** @brief retreive the succesor of the list element E */
12 struct lst_elm_t * getSuc(struct lst_elm_t * E);
13 /** @brief modify the date of the list element */
14 void setDatum(struct lst_elm_t * E, void * datum);
15 /** @brief modify the succesor of the list element */
16 void setSuc(struct lst_elm_t * E, struct lst_elm_t * S);
```

Apportez également les modifications nécessaires au type abstrait des *listes*:

```c
struct lst_t {
  struct lst_elm_t * head;
  struct lst_elm_t * tail;
  int numelm;
}

/** @brief Construire une liste vide */
struct lst_t * new_lst();
/** @brief Libèrer la mémoire occupée par la liste */
void del_lst(struct lst_t ** ptrL, void (*ptrFct) ());
/** @brief Vérifier si la liste L est vide ou pas */
bool empty_lst(const struct lst_t * L);
/** @brief Ajouter en tête de la liste L la donnée */
void cons(struct lst_t * L, void * datum);
/** @brief Ajouter en queue de la liste L la donnée */
void queue(struct lst_t * L, void * datum);
/** @brief Insérer dans la liste L la donnée suivant l'ordre donné par la fonction pointée par ptrFct */
void insert_ordered(struct lst_t * L, void * datum, bool (*ptrFct) ());
/** @brief Consulter la tête de la liste */
struct lst_elm_t * getHead(struct list * L);
/** @brief Consulter la queue de la liste */
struct lst_elm_t * getTail(struct list * L);
/** @brief Consulter le nombre d'éléments rangés dans la liste */
int getNumelm(struct list * L);
/** @brief Modifier le nombre d'éléments rangés dans la liste */
int setNumelm(struct list * L, int numElm);
/** @brief Visualiser les éléments de la liste L grâce à la fonction pointée par ptrFct */
void view_lst(struct lst_t * L, void (*ptrFct) () );
```

3 Pointeurs de fonctions

Grâce aux *pointeurs de fonctions* on peut définir les fonctions *génériques* de visualisation, de libération ou encore d'insertion ordonnée pour des listes *homogènes*. Et ce quelque soit le type *effectif* des `*data`[1].

3.1 Visualiser les données

Pour la visualisation de *listes de type entier*, les données (`datum`) sont des pointeurs d'entiers (`int *`) *transtypées*[2] en pointeurs de quelque choses (`void *`).

En définissant une fonction de visualisation pour un pointeur d'entiers :

```c
// fichier outils.c
void printInteger ( int * i ) {
  assert(i);
  printf( "La valeur est entière et vaut : %d\n", (*i) );
}
```

[1] `data` est un pointeur `void *` aussi `*datum` désigne la donnée proprement dite.

[2] *cast* in C.

et en la déclarant :

```
// fichier outils.h
void printInteger ( int * i );
```

on construit une *bibliothèque* `outils` qui sert à la visualisation des listes d'entiers :

```
// fichier lst.c
void print_lst(struct list * L, void (*ptrf) ()) {
  for (struct elmlist * E = L->head; E; E = E->suc ) {
    (*ptrf)(E->datum);
  }
}
```

où le paramètre `void (*ptrf) ()` est le pointeur de fonction *dont on n'a pas besoin de préciser les paramètres*.

Ensuite, on peut utiliser cette fonction de visualisation pour afficher les valeurs d'une liste d'entiers grâce à la fonction `printInteger` de notre boîte à *outils*:

```
print_lst(L, &PrintInteger);
```

On procède de la même manière pour visualiser les listes de réels, par exemple...

3.2 Libérer la mémoire

Rappel.
Lors de la libération de la mémoire allouée à une liste d'entiers la mémoire de tous les éléments était désallouée. La mémoire associées les valeurs entières était alors automatiquement *désallouée.*

Maintenant les éléments de liste connaissent non plus les données mais possèdent des pointeurs sur celles-ci. Désallouer la mémoire entraîne *obligatoirement* la désallocation de la mémoire des éléments de liste mais la mémoire allouée aux données proprement dites peut ou pas être désallouée. Ce choix est laissé à la fonction appelante grâce à un pointeur de fonction:

```
void del_lst(struct lst_t ** ptrL, void (*ptrFct) ())
```

où `ptrL` est un pointeur (de pointeur) de liste et `ptrFct` est un pointeur de fonction qui vaut:

- Soit `NULL` si la fonction appelante souhaite supprimer la liste *mais pas les données attachées*,
- Soit une référence à une fonction qui *permet de supprimer les données*

Pour les listes génériques d'entiers, il suffit d'ajouter aux outils la fonction suivante:

```c
// fichier outils.c
void freeInteger(int * i) {
  assert(i);
  free(i);
}
```

On procède de la même manière pour libérer les listes de réels, par exemple...

3.3 Travail dirigé

Servez vous des deux fonctions suivantes pour faire vos essais :

```c
// Déclarations et définitions à ajouter dans la fichier main.c
void listehomoreelle();
void listehomoentiere();

int main(){
  ...
}

void listehomoreelle() {
  struct list * L = new_lst();
  double * v;

  do {
    double u;

    printf ( "Entrez un réel (0 pour s'arrêter): " );
    scanf  ("%lf", &u );
    if ( u == 0 ) break;

    v = (double *)calloc(1, sizeof(double));
    *v = u;
    insert_ordered(&L, v, &cmpDouble);
  } while(true);
  view_lst(L, &printDouble);
  del_lst(&L, &rmDouble);
}

void listehomoentiere() {
  struct list * L = newLst();
  int * v;

  do {
    int u;

    printf ( "Entrez un entier (0 pour s'arrêter): " );
    scanf  ("%d", &u );
    if ( u == 0 ) break;

    v = (int *) calloc ( 1, sizeof(int) );
    *v = u;
    insert_ordered(&L,v,&cmpInteger);
  } while ( true );
  printLst( L, &printInteger);
  freeLst(&L, &rmInteger);
}
```

4 Quid des listes hétérogènes?

Modifiez la fonction principale `main` comme suit et vérifiez le bon fonctionnement:

```
int main() {
  int cmpt, a = 8, b = 4, * ptra = &a, * ptrb = &b;
  double x = 5.4, y = 3.14, * ptrx = &x, * ptry = &y;
  struct list * L = consVide();
  struct elmlist * iter;

  cons(L, ptra); // un entier
  cons(L, ptrb); // un entier
  cons(L, ptrx); // un réel
  cons(L, ptry); // un réel

  /* La liste vaut [3.14 ; 5.4 ; 4 ; 8 ] */
  for(cmpt = 0, E = getHead(L);
      cmpt < getNumelm(L);
      cmpt += 1, E = getSuc(E)) {
    if(cmpt < 2) {
      double * d = (double *) getData(E);
      printf( "La valeur est réelle et vaut : %lf\n", *d );
    } else {
      int * d = (int *) getData(E);
      printf( "La valeur est entière et vaut : %d\n", *d );
    }
  }
  return EXIT_SUCCESS;
}
```

4.1 Travail personnel

Modifiez les TA et à leurs fonctions membres pour que les types des données soient identifiables automatiquement.