Listes simplement chaînées génériques

S3B

Licence Informatique : Programmation Nécessaire

Stéfane

Table des Matières

1 Introduction 1

2 Listes génériques homogènes 1

3 Pointeurs de fonctions 2

Visualiser les données

Libérer la mémoire

Travail dirigé

4 Quid des listes hétérogènes? 4

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 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

```c
typedef struct list_elm {
  void * datum;
  struct list_elm * suc;
} list_elm_t;

list_elm_t * new_list_elm ( void * datum ); // Create a list element and store datum
void del_list_elm ( list_elm_t ** ptrE, void (*ptrFct) () ); // Delete list element

void * getDatum ( list_elm_t * E); // Retrieve the datum of the list element
void setDatum ( list_elm_t * E, void * datum ); // Modify the datum of the list element

list_elm_t * getSuc ( list_elm_t * E ); // Retrieve the successor of the list element
void setSuc ( list_elm_t * E, list_elm_t * S ); // Modify the successor of the list element
```

Modifiez le type abstrait des listes

```c
typedef struct list {
  list_elm_t * head;
  list_elm_t * tail;
  int numelm;
} list_t;

list_t * new_list ();/// @brief Construire une liste vide
void del_list ( list_t ** ptrL, void (*ptrFct) () );/// @brief Libèrer la mémoire occupée par la liste

bool empty_list ( const list_t * L );/// @brief Vérifier si la liste L est vide ou pas
list_elm_t * getHead ( list_t * L ); /// @brief Consulter la tête de la liste */
list_elm_t * getTail ( list_t * L ); /// @brief Consulter la queue de la liste */
void incNumelm ( list_t * L ); /// @brief Incrémenter de 1 le nombre d'éléments rangés dans la liste
void decNumelm ( list_t * L ); /// @brief Décrémenter de 1  le nombre d'éléments rangés dans la liste

void view_list ( list_t * L, void (*ptrFct) () ); /// @brief Visualiser les éléments de la liste L grâce à la fonction pointée par ptrFct

void cons( list_t * L, void * datum ); /** @brief Ajouter en tête de la liste L la donnée */
void queue ( list_t * L, void * datum ); /** @brief Ajouter en queue de la liste L la donnée */
void insert_ordered( list_t * L, void * datum, int (*ptrFct) () ); /// @brief Insérer dans L la datum suivant l'ordre donné par la fonction pointée par ptrFct
```

3 Pointeurs de fonctions

Grâce aux 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 effectivement pointé par `datum`[1].

3.1 Visualiser les données

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

Définissez une fct de visualisation pour un pointeur d'entiers dans `outils.c`

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

Déclarez la dans `outils.h`

```c
void printInteger ( int * i );
```

Utilisez la bibliothèque `outils` *pour visualiser des listes d'entiers*

```c
// fichier list.c
void view_list ( list_t * L, void (*ptrFct) () ) {
  for ( list_elm_t * E = L->head; E; E = E->suc ) {
    (*ptrFct) ( 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.

[1] `datum` *est un pointeur au format* `void *` *ce qui implique que* `*datum` *désigne la donnée proprement dite.*
[2] *cast in C.*

Grâce à la fonction `printInteger` de notre boîte à outils

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

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

3.2 Libérer la mémoire

Lors de la libération de la mémoire allouée à une liste d'entiers la mémoire de tous les éléments était désallouée. La mémoire allouée aux valeurs entières était alors automatiquement désallouée. Maintenant les éléments de liste connaissent non plus pas 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_list(struct list_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, les éléments de la listes mais pas les données attachées aux éléments,
- 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

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

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

3.3 Travail dirigé

Servez vous des deux fonctions suivantes pour faire vos essais

```
void listehomoentiere();
void listehomoreelle();
int main(){
  listehomoentiere();
  // listehomoreelle();
  return 0;
}
void listehomoentiere(){
  list_t * L = newLst();
  int * v;
  do{
    int u;
    printf( "Entrez un entier (0 pour s'arrêter): " );
    scanf( "%d", &u );
    if ( u == 0 ) break;
    v = calloc( 1, sizeof(int) );
    *v = u;
    cons( L, v ); // insert_ordered(L,v,&cmpInteger);
  } while ( true );
  view_list( L, &printInteger );
  del_list( &L, &rmInteger );
}
void listehomoreelle(){
  list_t * L = new_list();
  double * v;
  do{
    double u;
    printf( "Entrez un réel (0 pour s'arrêter): " );
    scanf( "%lf", &u );
    if ( u == 0 ) break;
    v = calloc( 1, sizeof(double) );
    *v = u;
    cons( L, v ); // insert_ordered(L,v,&cmpDouble);
  }while( true );
  view_list( L, &printDouble );
  del_list( &L, &rmDouble );
}
```

4 Quid des listes hétérogènes?

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

```
int main() {
  int a = 8, b = 4, * ptra = &a, * ptrb = &b;
  double x = 5.4, y = 3.14, * ptrx = &x, * ptry = &y;
  list_t * L = new_list();

  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 ]

  int cmpt;
  list_elm_t * E;
  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 0;
}
```