Série d'exercices #10

IFT-2035

November 28, 2023

10.1 Raisonner

Soit le morceau de code suivant, où `f` est une fonction quelconque que l'on ne connaît pas et où le langage utilise une syntaxe de style C:

```
{
  int table[2] = {0, 1};
  int size = 2;
  int tmp = 0;

  f (table, size);
  ...
}
```

On aimerait savoir si certaines conditions sont nécessairement toujours vraies aprés l'appel à `f`. On s'intéresse plus particulièrement aux conditions suivantes:

- `table[0] == 0`
- `size == 2`
- `tmp == 0`

Indiquer lesquelles de ces trois conditions sont nécessairement vraies dans chacun des cas suivants:

1. Le langage est exactement comme C: portée statique, passage d'arguments par valeur, affectation autorisée.
2. Le langage est comme C sauf que l'affectation (autre que l'initialization) est interdite.
3. Le langage est comme C sauf que les arguments sont passés par référence.
4. Le langage est comme C mais avec portée dynamique.

10.2 Gestion mémoire manuelle

Écrire une bibliothèque de listes simplement chaînées en C.

```
typedef struct list_elem list_elem;
struct list_elem {
  void *value;
  list_elem *next;
}
typedef struct list list;
struct list { list_elem *head; }

list *list_alloc  (void);
void  list_insert (list *l, void *v);
void *list_get    (list *l, int n);
...
```

1. Écrire le code des trois fonctions proposées.
2. Compléter en ajoutant les opérations suivantes: `list_remove`, `list_free`.
3. Décrire précisément les conditions nécessaires (que l'utilisateur de la bibliothèque doit suivre) pour qu'il n'y ait pas de déréférence de pointeur fou, ni de fuite.
4. Expliquer pourquoi ces conditions sont nécessaires et suffisantes pour garantir l'absence d'erreurs de type pointeur fou ou fuite.
5. Discuter de la flexibilité (ou de son manque) de ces conditions, vu du point de vue de l'utilisateur de cette bibliothèque.

10.3 Alignement et usage mémoire

Soit le code d'une bibliothèque C pour un *arbre binaire* qui associe des petits entiers de 16bit à une valeur en virgule flottante:

```
typedef struct bt_node bt_node;
struct bt_node {
  short    key;
  double   value;
  char     kind;
  bt_node *left;
  bt_node *right;
};
typedef struct bt bt;
struct bt { bt_node *root; };

bt  *bt_alloc  (void);
void bt_insert (bt *t, short k, double v);
char bt_remove (bt *t, short k);
bt  *bt_below  (bt *t, short k);
...
```

On sait aussi que sur le système qui nous intéresse, un `char` a une taille de 1B (byte), un `short` a une taille de 2B, que les pointeurs ont une taille de 4B, que les `double` ont une taille de 8B, et que tous ces types doivent obéir l'*alignement naturel*.

1. Quelle quantité de mémoire est allouée par un appel à `bt_alloc`?
2. Indiquer tous les endroits dans la structure `bt_node` où le compilateur C va insérer du *padding*.
3. Donner la valeur de `sizeof (bt_node)`.
4. La fonction `bt_below` ne modifie aucune partie de ses arguments et renvoie un tout nouvel arbre qui contient tous les éléments dont la clé est plus petite que `k`. Sachant que `t` a 10 nœuds, combien de bytes de mémoire doit-elle allouer dans le pire des cas?