Université BORDEAUX I

Année 1997-98
Session de janvier

Maîtrise d'informatique

Modèles et Techniques de Programmation (J.P.Braquelaire)

– CORRIGÉ DÉTAILLÉ –

Première partie

1°
- *(++ 3)*. Le symbole *++* est lié à une fonction. Par conséquent *(++ 3)* est un appel par valeur à cette fonction avec la valeur 3 en paramètre. Le corps de la fonction, c'est-à-dire l'expression *(setq x (+ 1 x))*, est évalué dans un contexte où le symbole *x* est lié à 3. Après cette évaluiation, le symbole *x* est, dans ce contexte, lié à 4 . Selon les implémentations, la valeur de l'affectation est soit 4, soit indéfinie.
- *(++ n)*. Même calcul avec la valeur 1 en paramètre. Le symbole *x* est lié à 2 dans le contexte local de l'appel. La valeur de *n* dans le contexte global n'est bien entendu pas modifiée.
- *(++ n)*. Idem.

2°
- *(++ 3)*. Le symbole *++* est lié à une macro. Par conséquent *(++ 3)* est une macro-expression évaluée en deux temps :
  a) Expansion de la macro expression. Cela donne

     (setq 3 (+ 1 3))

  b) Évaluation de la forme expansée. L'évaluation de l'expression ci-dessus tente de lier la constante entière 3 à la valeur 4, ce qui provoque une erreur.
- *(++ n)*. La forme expansée est lit(setq n (+ 1 n)). Elle est évaluée dans le contexte global, pour lequel *n* est un symbole défini lié à la valeur 1. Après évaluation, ce symbole est lié à la valeur 2. Comme précédemment, la valeur retournée est soit la valeur affectée (ici 2), soit indéfini.
-
- *(++ n)*. Comme précédemment avec *n* lié à 2 avant l'évaluation et à 3 après.

3° Cette question comportait un piège (involontaire, car je ne l'avais pas vu lorsque j'ai rédigé le sujet) : quel que soit le type de liaison (liaison lexicale ou dynamique) le résultat est le même. En effet, le résultat de l'expansion de *(let ((increment 5)) (++ n))* est :

(let ((increment 5)) (setq n (+ increment n)))

Lorsque cette expression est évaluée, la définition locale de *increment* (5) masque sa définition globale (1). Si l'on suppose qu'au début de l'évaluation de cette expression, le symbole global *n* est lié à la valeur 1, après cette évaluation il est lié à la valeur 6.

Lors de l'évaluation de *(++ n)*, la seule définition visible de *increment* — ici encore, quel que soit le mode d'évaluation — est la définition globale (1). Par conséquent, après évaluation, le symbole global *n* est lié à la valeur 7.

Deuxième partie

4° La fonction *qsort* implément un algorithme de tri indépendent du type des objets à trier. Elle utilise une description générique du type des objets, composée de la taille d'un objet et d'une fonction réalisant la comparaison de deux objets. La fonction *bsearch* est construite sur le même principe.

5° Si *point* est une classe implémentée en C standard, on peut par exemple faire l'hypothèse qu'il existe une fonction *point_input* de prototype

```
point point_input(void)
```

réalisant la saisie de coordonnées (clavier, souris...) et l'instanciation d'un point correspondant. Un exemple simple de code permettant de tester le mécanisme de la fonction *qsort* est :

```
point *point_array = malloc(number_of_points * sizeof(point));

for (i = 0; i < number_of_points; i++)
    point_array[i] = point_input();

qsort(point_array, number_of_points, sizeof(point), point_compare);
```

Il y avait bien entendu d'autres solutions.[1]

6° Une solution simple, analogue à la solution présentée en cours d'extension de la fonction *tgetent* (bibliothèque GNU Termcap) consistait à permettre deux types d'appel pour *bsearch* :

a) *bsearch(key, array, ...)* pour rechercher la première occurence de *key* dans le vecteur *array* ;

b) *bsearch(NULL, array, ...)* pour rechercher l'occurence suivante de *key*. Cet appel peut être itéré jusqu'à ce que *bsearch* retourne *NULL*.

De cette façon, le comportement standard de *bsearch* n'est pas altéré et l'on préserve la compatibilité ascendante.[2]

7° – Interface (minimale) :

```
# ifndef ARRAY_H
# define ARRAY_H

typedef struct array *array;
typedef int (*array_sort_fun)(const void *, const void *);

array array_make(array_sort_fun compare);
void array_qsort(array self);
void *array_bsearch(array self, const void *key);

void array_set(array self, int rank, void *element);
```

1. On pouvait encore faire l'hypothèse d'une fonction d'instanciation *point_make* recevant en paramètre les coordonnées d'un point à instancier et lire ou générer aléatoirement ces coordonnées. Cependant, cela nécessitait d'effectuer des hypothèse sur la dimension des points et le type des coordonnées. Il ne fallait par contre en aucun cas travailler directement sur l'implémentation d'un objet de la classe *point*.

2. La solution consistant à itérer *bsearch(key, array, ...)* jusqu'à ce que la fonction retourne *NULL* constitue une modification du comportement standard de *bsearch* et ne préserve pas la compatibilité. La solution consistant à retourner un vecteur avec la suite de toutes les références identiques pose, entre autre, un problème de gestion de la mémoire. En effet une fonction utilisant *bsearch* de manière standard n'aura aucune raison de libérer ces vecteurs et l'on aura des fuites mémoires.

```
void *array_get(array self, int rank);

# endif /* ARRAY_H */
```

La fonction de comparaison associée aux élément du vecteur est fournie lors de l'instanciation de chaque vecteur. Elle est mémorisée dans une variable d'instance

– Implémentation des fonction `array_qsort` et `array_bsearch` : une instance de la classe `array` doit contenir les information nécessaires à l'implémentation du vecteur, par exemple les attributs :
  – `area` référençant le début du vecteur ;
  – `size` indiquant le nombre d'éléments du vecteur ;
  – `compare_fun` référençant la fonction de comparaison associée aux éléments du vecteur.

```
void array_qsort(array self)
{
    qsort(self->area, self->size, sizeof(void *), self->compare_fun);
}
void *array_bsearch(array self, const void *key)
{
    return bsearch(key, self->area, self->size, sizeof(void *),
                   self->compare_fun);
}
```

8° Modification de l'exemple :

```
array point_array = array_make(point_compare);

for (i = 0; i < number_of_points; i++)
    array_set(point_array, i, point_input());

array_qsort(point_array);
```