Correction de l'examen final de programmation impérative

ensIIE, semestre 1

mercredi 21 janvier 2026

Remarques préliminaires

— Il ne sert à rien de recopier les informations de type dans les `@requires` (exemple `@requires n est un entier`), puisqu'elles sont déjà dans le prototype de la fonction.

— Si on affecte un pointeur avec une valeur, il ne faut pas faire une allocation mémoire quand on l'initialise, sinon on a une fuite mémoire.

Exemple à ne pas suivre :

```
void print_list(list l) {
  list v = malloc(sizeof (struct bucket));
  v = l;
  while (v != NULL) {
    printf("%g␣->␣", v->val);
    v = v->next;
  }
  printf("[]");
}
```

— Les accolades ne sont obligatoires après un **`if`**, un **`else`** ou un **`while`** que s'il y a plusieurs instructions concernées.

Exercice 1 : Fonctions simples (6 points)

1. On rappelle que **`char`** est un type d'*entiers* sur 1 octet. Ces entiers peuvent être vus comme des caractères d'impression via le code ASCII, mais on peut travailler avec eux comme des entiers usuels (modulo 2^8).

   Il faut passer la variable par référence.

   ```
   /*@requires a is a valid address
     @assigns *a
     @ensures *a is increased by 1 */
   void incr(char *a) {
     *a += 1;
   }
   ```

2. Remarque : `@requires p is a valid pair` n'a aucun sens : un **`struct pair`** est un enregistrement et pas une adresse ; au pire il peut ne pas avoir été initialisé, mais cela n'empêche pas la fonction d'être correcte.

```
/*@requires n > 0
  @assigns nothing
  @ensures an array of size n, containing copies of p, is returned */
struct pair *make_array(int n, struct pair p) {
  struct pair *r = malloc(n * sizeof (struct pair));
  for (int i = 0; i < n; i += 1)
    r[i] = p;
  return r;
}
```

Si on voulait ne pas mettre de **struct** partout, il fallait faire la déclaration de type

```
typedef struct pair pair;
```

3. On rappelle qu'en C, la valeur fausse est représentée par 0, et la valeur vraie par n'importe quel autre entier (en général 1).

```
/*@requires l is a well-formed acyclic list
  @assigns nothing
  @ensures 1 is returned if the length of l is even, 0 otherwise */
int even_length(list l) {
  int c = 0;
  while (l != NULL) {
    c += 1;
    l = l->next;
  }
  return c % 2 == 0;
}
```

On évitera les `if (condition) return 1; else return 0;` auquel on préférera `return condition;`.
On pouvait aussi faire :

```
int even_length_clever(list l) {
  while (l != NULL && l->next != NULL)
    l = l->next->next;
  return l == NULL;
}
```

4.
```
/*@requires s1 and s2 are nul-terminated strings
  @assigns nothing
  @ensures The number of occurrences of s1 in s2 is returned */
int count_sub(char *s1, char *s2) {
  int i = 0;
  int j = 0;
  int c = 0;
  /*@loop invariants
    The substrings of s1 between position 0 and j - 1
    and s2 between positions i and i + j - i are equal.
    There are c occurrences of s1 in s2 starting at positions
    between 0 and i - 1. */
```

```
  while (s2[i+j] != '\0')
    if (s1[j] == '\0') {
      /* We reached the end of s1, an occurrence of s1 in s2 has
         been found at position i */
      c += 1;
      i += 1; /* We restart at position i + 1 */
      j = 0;
    } else if (s1[j] == s2[i+j])
      j += 1; /* s1 and s2 from position i coincide up to j */
    else {
      /* s1 and the substring of s2 starting at i differ.
         We restart at position i + 1 */
      i += 1;
      j = 0;
    }
  /* If we reach the end of s2, there is no point in looking at
     starting positions after i */
  if (s1[j] == '\0') c += 1; // Do not forget the last position
  return c;
}
```

Exercice 2 : Jeu des 5 erreurs (5 points)

1. Rappel : le type **double** est celui des nombres à virgule flottante en double précision.
   Ici, il n'y a pas à véritablement parler d'erreur de type mais plutôt un problème d'arrondi : dans l'expression `sum / s`, comme `sum` et `s` sont tous deux de type **int**, la division se fait sur les entiers. On aura donc uniquement la partie entière du quotient de `sum` par `s`. Le résultat est ensuite converti implicitement en **double**.
   Pour être certain que la division se fasse sur des **double**, on peut soit forcer la conversion d'un des deux côtés : `(double) sum / s` ou `sum / (double) s`, ou alors on peut directement déclarer `sum` comme un **double**.
   Remarque : la "solution" suivante ne fonctionne pas puisqu'on aura le même souci de division entière :
   ```
   double res = sum / s;
   return res;
   ```
2. Le problème ici vient de la taille allouée par le `malloc` : on doit réserver la taille d'un maillon et pas uniquement d'une `list`, qui est un pointeur vers un maillon. Il faut donc écrire `malloc(sizeof (struct bucket))`.
3. Le problème est dans la condition du **if** : c'est une affectation (`=`) et non une comparaison (`==`). La valeur de `*l` est donc remplacée par `NULL`, et la condition est évaluée en `NULL`, donc en `0`, donc en la valeur fausse. On entre donc systématiquement dans le case **else** et on a une erreur de segmentation au premier `v->next` puisque `v` vaut `NULL`.
4. Premièrement, `r` n'est pas initialisé. Sa valeur est donc indéterminée et ne vaut donc pas nécessairement `0` au départ.

Deuxièmement, il manque des accolades après le **`while`**. Par conséquent, seule l'instruction `r += 1` fait partie du corps de la boucle, et donc, dans le cas où `l` n'est pas la liste vide, la boucle ne terminera jamais.

Exercice 3 : Représentation de relations (9 points)

1.
```
/*@requires nothing
  @assigns nothing
  @ensures a representation of the empty relation is returned */
list *empty_rel() {
  list *r = malloc(N * sizeof (list));
  for (int i = 0; i < N; i += 1)
    r[i] = NULL;
  return r;
}
```

2.
```
/*@requires 0 <= k < N
  @assigns m[k]
  @ensures The association of k to v is added to the relation m */
void add_to_rel(key k, value v, list *m) {
  list a = malloc(sizeof (struct bucket));
  a->val = v;
  a->next = m[k];
  m[k] = a;
}
```

   Il n'est pas nécessaire de passer `m` encore plus par référence, puisque c'est un tableau.

3.
```
/*@requires 0 <= k < N
  @assigns nothing
  @ensures the values associated to k in m are printed */
void print_values(key k, list *m) {
  printf("%d␣->␣{␣", k);
  list l = m[k];
  while (l != NULL) {
    printf("%g␣", l->val);
    l = l->next;
  }
  printf("}\n");
}
```

4.
```
/*@requires 0 <= k < N
  @assigns m[k] and its buckets
  @ensures all values associated to k in m are removed */
void remove_all(key k, list *m) {
  list l = m[k];
  while (l != NULL) {
```

```
    list tmp = l;
    l = l->next;
    free(tmp);
  }
  m[k] = NULL;
}
```

5.
```
/*@requires nothing
  @assigns nothing
  @ensures a representation of the empty relation is returned */
struct cell *empty_rel() {
  struct cell *r = malloc(N * sizeof (struct cell));
  for (int i = 0; i < N; i += 1)
    r[i].key = -1;
  return r;
}
```

6.
```
/*@requires m has size N
  @assigns nothing
  @ensures The address of an free cell in m is returned
    or NULL if no such free cell exists */
struct cell *get_free_space(struct cell *m) {
    int i = 0;
    while (i < N && m[i].key != -1)
      i += 1;
    if (i == N) {
      fprintf(stderr, "No␣more␣space!\n");
      return NULL;
    }
    return &m[i];
}
```

7.
```
/*@requires 0 <= k < N
  @assigns nothing
  @ensures the values associated to k in m are printed */
void print_values(key k, struct cell *m) {
  printf("%d␣->␣{␣", k);
  if (m[k].key == k) {
    cell_list l = &m[k];
    while (l != NULL) {
      assert (l->key == k);
      printf("%g␣", l->val);
      l = l->next;
    }
  }
  printf("}\n");
}
```

8.
```
/*@requires 0 <= k < N
  @assigns m[0] ... m[N-1]
  @ensures all values associated to k in m are removed */
void remove_all(key k, struct cell *m) {
  if (m[k].key != k) return;
  cell_list l = &m[k];
  while (l != NULL) {
    assert(l->key == k);
    l->key = -1;
    l = l->next;
  }
}
```

9.
```
/*@requires 0 <= k < N
  @assigns m[0] ... m[N-1]
  @ensures The association of k to v is added to the relation m */
void add_to_rel(key k, value v, struct cell *m) {
  if (m[k].key == -1) {
    m[k].key = k;
    m[k].val = v;
    m[k].next = NULL;
  }
  else if (m[k].key == k) {
    struct cell *f = get_free_space(m);
    if (f == NULL) return;
    f->key = k;
    f->val = v;
    f->next = m[k].next;
    m[k].next = f;
  } else {
    struct cell *f = get_free_space(m);
    if (f == NULL) return;
    cell_list prec = &m[m[k].key];
    while (prec->next != NULL && prec->next != &m[k])
      prec=prec->next;
    assert(prec->next != NULL);
    f->key = m[k].key;
    f->val = m[k].val;
    f->next = m[k].next;
    prec->next = f;
    m[k].key = k;
    m[k].val = v;
    m[k].next = NULL;
  }
}
```