Solutions: Série 6 - Pointeurs - ICC-C 2025-2026

1. Lecture et compréhension

Le premier code affiche

```
v = 5
plus 10 = 15
plus 10 = 25
v = 5
```

Après le premier affichage de `v = 5`, on appelle `plus10(v)`. Cela recopie la valeur `5` dans le paramètre `a` de la fonction `plus10`. On ajoute `10` à `a` puis on l'affiche, ce qui donne 15. Cette même valeur 15 est renvoyée en valeur de retour. Elle est donc recopiée pour le paramètre `a` d'un nouvel appel à `plus10`. Ce deuxième appel affiche puis retourne 25.

La valeur de retour est ensuite *ignorée*. On n'en fait plus rien. À aucun moment, on n'a modifié `v`, donc la dernière instruction de `main` affiche à nouveau `v = 5`.

Le second code, par contre, affiche

```
v = 5
plus 10 = 15
plus 10 = 25
v = 25
```

Cette fois, ce n'est pas la *valeur* `5` qui est transmise à `ptr_plus10`. On transmet et recopie *l'adresse* de la variable `v`. Mais on ne recopie pas la *valeur contenue* dans cette variable.

Lorsque le premier appel à `ptr_plus10` manipule `*a`, il manipule la *même case mémoire* que le `v` original. On augmente donc le `v` original à 15, et on l'affiche.

Là aussi, on renvoie toujours `a`, qui est bien toujours l'adresse de `v`. On recopie cette adresse (mais pas la valeur) dans le nouveau paramètre `a` du second appel à `ptr_plus10`. Ce second appel manipule donc toujours la même case mémoire ; celle de `v`.

On ignore toujours le résultat du second appel, qui est l'adresse. Mais entre temps, on a bel et bien modifié la valeur contenue dans la variable `v`. C'est pourquoi la dernière instruction affiche `v = 25`.

2. Échauffement : échanger trois variables

```c
void cycle(int *a, int *b, int *c) {
  int old_a = *a; // à sauvegarder d'abord, avant de faire *a = ...
  *a = *b;
  *b = *c;
  *c = old_a;
}

int main() {
  int x = 5, y = 7, z = 17;
  cycle(&x, &y, &z);
  printf("%d %d %d\n", x, y, z);

  return 0;
}
```

3. Stack (pile)

```c
#include <stdio.h>
#include <stdbool.h>

bool push(int stack[], size_t *stack_size, size_t max_stack_size, int elem) {
  if (*stack_size < max_stack_size) {
    // On a encore de la place ; on peut empiler l'élément
    stack[*stack_size] = elem;
    *stack_size += 1; // ou (*stack_size)++; mais c'est un peu obscur
    return true;
  } else {
    // Il n'y a plus de place ; on ne touche à rien et on renvoie false
    return false;
  }
}

bool pop(int stack[], size_t *stack_size, int *elem) {
  if (*stack_size > 0) {
    // Il y a au moins un élément dans la pile
    *stack_size -= 1; // ici on décrémente d'abord stack_size
    *elem = stack[*stack_size];
    return true;
  } else {
    // La pile est vide ; on ne touche à rien et on renvoie false
    return false;
  }
}

bool equalstr(const char s1[], const char s2[]) {
  size_t i = 0;
  while (s1[i] != '\0' && s2[i] != '\0') {
    if (s1[i] != s2[i]) {
      return false;
    }
    i++;
  }

  /* Si on est arrivé au bout d'une des chaînes, alors on vérifie qu'on est
   * bien aussi au bout de l'autre chaîne.
   */
  return s1[i] == '\0' && s2[i] == '\0';
}
```

```c
int main() {
  /* D'abord lire la première ligne, qui contient notamment la hauteur maximale
   * de la pile.
   */
  size_t max_stack_size;
  scanf("%lu", &max_stack_size);
  size_t instruction_count;
  scanf("%lu", &instruction_count);

  // On peut maintenant créer la pile elle-même
  int stack[max_stack_size];
  size_t stack_size = 0;

  // Maintenant on lit les instructions
  for (size_t i = 0; i < instruction_count; i++) {
    // Lire le premier mot
    char instruction[100];
    scanf("%s", instruction); // pas de & ici car on lit une chaîne

    if (equalstr(instruction, "push")) {
      // Aussi lire l'élément à ajouter à la pile
      int elem;
      scanf("%d", &elem);

      // Tenter de l'ajouter ; en cas d'échec, afficher le message demandé
      if (!push(stack, &stack_size, max_stack_size, elem)) {
        printf("Étape %lu : pile remplie\n", i);
      }
    } else if (equalstr(instruction, "pop")) {
      // Tenter de retirer un élément
      int elem;
      if (pop(stack, &stack_size, &elem)) {
        printf("%d\n", elem);
      } else {
        printf("Étape %lu : pile vide\n", i);
      }
    } else {
      // Uh oh ; on ne connaît pas cette instruction (pas demandé dans l'énoncé)
      printf("Étape %lu : instruction inconnue '%s'\n", i, instruction);
    }
  }

  return 0;
}
```

4. « The floor is lava »

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <assert.h>

/* Retire le dernier caractère de `str`, qu'on suppose être `\n` */
void remove_trailing_newline(char str[]) {
  size_t len = strlen(str);
  assert(len > 0);
  str[len - 1] = '\0';
}

/* Lit le contenu de la grille depuis l'entrée. */
void read_grid_content(bool grid[], size_t M, size_t N) {
  for (size_t l = 0; l < M; l++) {
    for (size_t c = 0; c < N; c++) {
      int cell;
      scanf("%d", &cell);
      grid[l*N + c] = (cell != 0); // calcul ligne * nombre_colonnes + colonne
    }
  }
}

/* Exécute une instruction.
 * - grid de taille MxN
 * - (*L, *C) est la position de Bix, avant et après le déplacement
 * - `move` est le caractère du déplacement à effectuer
 */
void execute_one_move(const bool grid[], size_t M, size_t N,
    size_t *L, size_t *C, char move) {

  /* Déplacer d'une case dans la direction indiquée par `move`.
   * Ne pas bouger si cela devait faire sortir Bix de la grille.
   */
  if (move == 'h') {
    if (*L > 0) {
      *L -= 1;
    }
  } else if (move == 'b') {
    if (*L < M-1) {
      *L += 1;
    }
  } else if (move == 'g') {
    if (*C > 0) {
      *C -= 1;
    }
  } else if (move == 'd') {
    if (*C < N-1) {
      *C += 1;
    }
  }
}
```

```c
/* Exécute la séquence d'instructions.
 * - grid de taille MxN
 * - (*L, *C) est la position de Bix, avant et après les instructions
 * - moves est le "mot" des déplacements
 *
 * Renvoie l'indice de l'instruction où l'algorithme s'est arrêté. Si c'est
 * égal à strlen(moves), Bix n'est pas tombé dans la lave.
 */
size_t execute_moves(const bool grid[], size_t M, size_t N,
    size_t *L, size_t *C, const char moves[]) {

  size_t i = 0;
  while (moves[i] != '\0') {
    execute_one_move(grid, M, N, L, C, moves[i]);

    // Vérifier si Bix est tombé dans la lave
    // Rappel : les cases de lave sont celles où la valeur est 0/false
    if (!grid[(*L) * N + (*C)]) {
      return i;
    }

    i++;
  }

  return i;
}
```

```c
int main() {
  /* On peut lire plus d'une variable avec scanf. Ce n'est pas toujours
   * forcément très clair, mais ici pourquoi pas ?
   */
  size_t M, N;
  scanf("%lu %lu", &M, &N);

  // Créer le tableau représentant le champ. On a besoin de M x N cases.
  size_t grid_size = M * N;
  bool grid[grid_size];
  read_grid_content(grid, M, N);

  // Lire la séquence de déplacement
  char moves[100];
  fgets(moves, 100, stdin); // un fgets "pour rien" pour finir la ligne de scanf
  fgets(moves, 100, stdin);
  remove_trailing_newline(moves);

  // Bix commence en (0, 0)
  size_t L = 0;
  size_t C = 0;

  // Exécuter les déplacements
  size_t final_instr = execute_moves(grid, M, N, &L, &C, moves);

  // Afficher le bon message en fonction de la situation de fin
  if (moves[final_instr] != '\0') {
    // final_instr ne correspond pas à strlen, donc Bix est tombé dans la lave
    printf("Game over %lu. Bix est tombé à (%lu, %lu)\n", final_instr, L, C);
  } else if ((L == M - 1) && (C == N - 1)) {
    // Bix s'est arrêté sur la case de sortie !
    printf("Bravo Bix !\n");
  } else {
    // Bix n'est pas tombé dans la lave mais ne s'est pas arrêté sur la sortie.
    printf("Bix se trouve à (%lu, %lu)\n", L, C);
  }

  return 0;
}
```