3TC31 (ex. INF107)

Examen intermédiaire (deuxième partie)
Éléments de correction

2025–2026

Exercice 1 : Questions de cours (4 points / 4 minutes)

Question 1 (2 points)

— Variable locale (*Automatic Storage Duration*) - La pile (*Stack Data*) : les variables locales d'une fonction sont stockées dans la pile
— Variable globale (*Static Storage Duration*) - Données globales (*Global Data*) : les variables globales sont stockées dans la zone des données globales (`data`, `rodata` ou `bss`)

Question 2 (2 points)

```
1 int main(int argc, char *argv[]) {
2   int a = 4;
3   int *b = &a;
4   *b = 5;
5   return EXIT_SUCCESS;
6 }
```

Juste avant la fin du programme, la variable `a` contient la valeur 5 et la variable `b` contient l'adresse de la variable `a`.

Ligne 2, le programme déclare une variable (locale) `a`, de type `int` initialisée à la valeur 4. Ligne 3, le programme déclare une variable (locale) `b`, de type `int *` (c'est-à-dire pointeur vers un `int`), initialisée avec l'adresse de la variable `a` (adresse obtenue grâce à l'opérateur `&`). Ligne 4, on déréférence le pointeur `b` grâce à la construction `*b`, c'est-à-dire que l'on considère le contenu de `b` comme une adresse mémoire (en l'occurrence le contenu de `b` est l'adresse de la variable `a`), et on va écrire à cette adresse la valeur 5. La valeur 5 est donc écrite à l'adresse contenue dans `b`, c'est-à-dire là où se situe la variable `a`. La variable `a` prend donc la valeur 5.

Exercice 2 : Questions de cours (16 points / 26 minutes)

Question 3 (2 points)

```
enum kind {ZERO, ONE, NOT, XOR, REG};
typedef enum kind kind_t;
```

Il est possible d'écrire de manière plus concise :

```
typedef enum {ZERO, ONE, NOT, XOR, REG} kind_t;
```

Question 4 (4 points)

```
struct circuit {
  kind_t kind;
  _Bool value, next_value;
  struct circuit *inputs[2];
};

typedef struct circuit circuit_t;
```

Question 5 (4 points)

```
_Bool step_combinational(circuit_t *c) {
  switch(c->kind)
  {
    case ZERO:
      return 0;
    case ONE:
      return 1;
    case REG:
      return c->value;
    case NOT:
      return !step_combinational(c->inputs[0]);
    case XOR:
      return step_combinational(c->inputs[0]) ^ step_combinational(c->inputs[1]);
  }
  return 0;
}
```

La fonction récupère un pointeur vers le circuit (`circuit_t *`). Donc pour accéder à ses membres (`kind`...), il faut déréférencer ce pointeur `c`, soit explicitement via la construction `(*c).kind` (les parenthèses sont obligatoires à cause des priorités respectives des opérateurs `*` et `.`), soit implicitement avec la construction `c->kind`.

Question 6 (2 points)

```
void print_regs(circuit_t c[], unsigned int n) {
  for(unsigned int i = 0; i < n; i++) {
    if (c[i].kind == REG)
      printf("%c", c[i].value ? 't' : 'f');
  }
  printf("\n");
}
```

L'opérateur `? :` est un opérateur ternaire (qui prend trois opérandes) : par exemple `a ? b : c`. Si `a` est évalué à une valeur différente de 0, l'opérande `b` est évalué et le résultat de l'expression `a ? b : c` est égal au résultat de l'évaluation de `b`. Dans le cas contraire, l'opérande `c` est évalué et le résultat de l'expression `a ? b : c` est égal au résultat de l'évaluation de `c`.

Donc ici, si `c[i].value` vaut `true` (différent de 0), `(c[i].value ? 't' : 'f')` vaut `'t'`. Dans le cas contraire, l'expression `(c[i].value ? 't' : 'f')` vaut `'f'`.

Il ne reste plus qu'à afficher le caractère, ce qui est faisable avec la chaîne de format `"%c"` passée en premier argument de `printf`. Pour rappel, `'t'` est un entier égal au code ASCII correspondant au caractère `t` (116). `"%c"` indique à `printf` que l'argument est un entier et qu'il doit afficher le caractère correspondant (donc ici `t`).

Question 7 (4 points)

```
1  int main(int argc, char *argv[]) {
2    circuit_t *c = malloc(4 * sizeof(circuit_t));
3    if (!c) // Ou (c == NULL)
4    {
5      perror("malloc␣failed:");
6      return EXIT_FAILURE;
7    }
8
9    // Create 4 circuit elements for a simple 2-bit counter:
10   circuit_t *not = &c[0], *xor = &c[1], *reg0 = &c[2], *reg1 = &c[3];
11   mk_not(not , reg0);
12   mk_xor(xor , reg0, reg1);
13   mk_reg(reg0, not);
14   mk_reg(reg1, xor);
15
16   // Simulate the 4 circuit elements for 5 clock ticks:
```

```
17    for(unsigned int i = 0; i < 5; i++) {
18      print_regs(c, 4);
19      step_circuit(c, 4);
20    }
21
22    free(c);
23
24    return EXIT_SUCCESS;
25  }
```

La ligne 10 nous permet de conclure que le tableau à allouer doit pouvoir contenir au moins 4 structures `circuit_t`. L'allocation se fait à l'aide de la fonction `malloc` en lui passant le nombre d'octets à allouer, donc ici 4 fois la taille de la structure `circuit_t` (d'où `4 * sizeof(circuit_t)`). `malloc` renvoie l'adresse du premier octet de la zone allouée ou la valeur `NULL` en cas d'erreur. Ce dernier cas est testé par le `if` ligne 3.

Une fois que la mémoire allouée n'est plus utilisée, il faut la libérer à l'aide de la fonction `free`, à qui on doit passer le pointeur renvoyé par la fonction `malloc`.