Examen, 2h, avec documents
Corrigé

NOM : **Prénom :**

Consignes : Téléphones, ordinateurs et calculatrices interdits.
Documents de cours/TD/TP autorisés.
Aucune question possible : en cas d'ambiguïté dans le sujet, signaler le choix fait sur la copie.

Conseil : Lire complètement le sujet avant de commencer à y répondre.

Barème indicatif :

exercice	1	2	3
points	5	5	10

Exercice 1 Répondre de manière concise et précise aux questions suivantes.
1. On considère le sous-programme suivant :

```
1  int * sp() {
2      int tab[10];
3      int i = 0;
4      while (i <= 10) {
5          tab[i] = i * i;
6          i++;
7      }
8      return tab;
9  }
```

Il contient au moins deux maladresses et deux erreurs. Lesquelles ? Justifier les réponses et corriger le programme.

Solution :

```
1  #include <stdlib.h>
2  #define MAX 10  // maladresse 1
3  int * sp() {
4      int* tab = malloc(MAX * sizeof(int));       // Erreur 2
5      for (int i = 0; i < MAX; i++) {     // maladresse 2 : for et non while
6          // Erreur 1 < et non <=
7          tab[i] = i * i;
8      }
9      return tab; // Erreur 2
10 }
```

Maladresse 1 : Ne pas utiliser une constante littérale (10) mais symbolique (MAX).
Maladresse 2 : Ne pas utiliser un `while` là où un `for` s'applique !

Erreur 1 : Mauvais indice. 10 n'est pas valide car on commence à 0.

Erreur 2 : On ne doit pas retourner l'adresse d'une variable locale car cette mémoire sera libérée quand le sous-programme se terminera. Ici on a choisi d'allouer dynamiquement la mémoire du tableau. L'appelant devra donc la libérer. Une autre solution aurait été d'avoir le tableau en paramètre du sous-programme ainsi que sa capacité.

2. Expliquer la différence entre taille effective et capacité d'un tableau.

Solution : La capacité correspond à la quantité de mémoire réservée pour le tableau. La taille effective est le nombre d'éléments effectivement stockés dans le tableau, i.e. la partie utilisée du tableau. La capacité comme la valeur maximale de la taille effective.

3. Le terme sous-programme est le terme général qui désigne les procédures et les fonctions.

3.1. Indiquer l'intérêt d'un sous-programme.

Solution : Permettre au programmeur de définir ses propres instructions et expressions et ainsi factoriser et réutiliser du code mais aussi structurer ses programmes.

3.2. Expliquer la différence entre procédure et fonction.

Solution : Une procédure correspond à une instruction et n'a donc pas de valeur.

Une fonction correspond à une expression et a donc une valeur.

3.3. À quoi reconnait-on une procédure en C ?

Solution : C'est une fonction dont le type de retour est `void`.

4. Expliquer les notions de *paramètre formel* et de *paramètre effectif*.

Solution : Un paramètre formel est utilisé dans un sous-programme pour désigner une information qui sera échangée avec le programme appelant. Le paramètre effectif est donné lors de l'appel du sous-programme et sert à initialiser le paramètre formel correspondant.

Exercice 2 : Spécifier des sous-programmes

Pour chacun des énoncés suivants, donner la spécification du sous-programme correspondant.

1. Calculer la puissance entière d'un réel

   Solution :

   (a) **Objectif :** La puissance entière d'un réel

   (b) **Exemples :**
   - nominal puissance positive : $2^4 = 16$
   - nominal puissance nulle $5^0 = 1$
   - nominal puissance négative $2^{-3} = 0.125$
   - nominal puissance négative, nombre négatif $(-2)^{-3} = -0.125$
   - nominal avec un vrai nombre réel $(-1.2)^2 = 1.44$
   - nominal nombre négatif, puissance négative $(-3)^3 = -27$

   Remarque : ici je n'ai pas mis systématiquement des réels car en math en ne le fait pas. En Ada, il faudra le faire.

   Ceci permet de bien identifier les informations fournies au sous-programme : le nombre et l'exposant et les informations attendues la puissance.

   (c) **Paramètres :** (on identifie d'abord le rôle, le mode, le type puis le nom)
   - nombre : in Réel – – nombre réel

— exposant : in Entier – – l'exposant
— puissance : out Réel – – la puissance (nombre ** exposant)

(d) **Type de sous-programme :** Fonction car un seul paramètre en sortie et les autres en entrée.

Remarque : le nom du paramètre en sortie est un bon candidat pour le nom de la fonction !

(e) **Préconditions :** Non (x = 0 Et n <= 0) <=> x /= 0 Ou n > 0

(f) **Postcondition :** puissance == nombre ** exposant = Produit(nombre) n fois si n > 0...

```
-- La puissance entière d'un nombre réel
-- Paramètres
--      - nombre   : in  Réel   -- le nombre réel
--      - exposant : in  Entier -- l'exposant
-- Retourne
--      - puissance : out Réel  -- nombre à la puissance exposant
--
-- Nécessite  : x /= 0 Ou n > 0
-- Assure : pas si simple à exprimer !
-- Exemples :
--      On peut reprendre quelques uns de ceux qui sont donnés avant
Fonction Puissance (Nombre : in Réel; exposant : in Entier) retourne Réel
```

La signature en C est :

```
double puissance(double nombre, int exposant);
```

2. Saisir un entier compris entre une borne inférieure et une borne supérieure. Avant chaque demande à l'utilisateur, une consigne lui est affichée.

   Solution :

   (a) **Objectif :** L'énoncé ci-dessus.

   (b) **Exemples :**
   - nominal un seul essai inf = 10, sup = 15, utilisateur : 10. Nombre = 12.
   - nominal trop petit : inf = 10, sup = 15, utilisateur 9, 16, 13. Nombre = 13
   - limite : borne inférieure choisie : inf = 10, sup = 15, utilisateur 10. Nombre = 10
   - limite : borne supérieure choisie : inf = 10, sup = 15, utilisateur 15. Nombre = 15
   -

   On pourrait ajouter la consigne et montrer précisément ce qui doit se passer. Ici l'idée est plutôt de spécifier l'interface du programme avec son utilisateur, en particulier en précisant les messages.

   (c) **Paramètres :**
   - inf : in Entier – borne inférieure
   - sup : in Entier – borne supérieure
   - consigne : in Chaine – le message à afficher à l'utilisateur
   - nombre : out Entier – l'entier saisit par l'utilisateur

   (d) **Type de sous-programme :** on choisit de faire une procédure car appelé avec les mêmes entrées on pourra avoir un résultat différent.

   Le fait que l'on fasse des E/S est aussi une indication d'une procédure.

   (e) **Précondition :** inf <= sup

   (f) **Question :** faut-il prévoir une contrainte sur le message ?

   (g) **Postcondition :** inf <= nombre Et nombre <= sup

```
-- Saisir une entier compris entre une borne inférieure et une borne
-- supérieure.  Un message d'afficher la consigne à l'utilisateur.
--
-- Paramètre :
--      Inf, Sup: in Entier  -- borne dans lesquelles doit être l'entier saisi
```

```
--      Consigne: in Chaine  -- le message à afficher à l'utilisateur
--
-- Nécessite : Inf <= Sup
-- Assure :  Inf <= Nombre Et Nombre <= Sup
-- Exemple : Saisir
Procédure Saisir(Nombre : out Entier ; Inf, Sup: in Entier ; Consigne: in Chaine);
```

La signature en C est :

```
void saisir_entier(int *entier, inf, int sup, const char consigne[]);
```

3. Savoir si une année est bissextile

   Solution :

   Remarque : Il faudrait suivre la même démarche que dans les deux premières questions. Ici je ne donne que le résultat.

```
-- Est-ce qu'une année est bissextile ?
--
-- Paramètres :
--     - Année : in Entier
-- Retourne Booléen
--
-- Nécessite : Année > 0
-- Assure : (Année Mod 4 = 0) Et ((Année Mod 100 /= 0) Ou (Année Mod 400 = 0))
-- Exemples :
--      2017 --> FAUX -- n'est pas bissextile
--      2016 --> VRAI -- est bissextile
--      2020 --> VRAI -- est bissextile
--      1900 --> FAUX -- n'est pas bissextile
--      2000 --> VRAI -- est bissextile
Fonction Est_Bissextile (Année : in Entier) retourne Booléen
```

   La signature en C est :

```
bool est_bissextile(int annee);
```

4. Calculer le pgcd de deux entiers strictement positifs

   Solution :

```
-- Le pgcd de deux entiers strictement positifs.
--
-- Paramètres :
--      a, b: in Entier
-- Retourne : Entier
--
-- Nécessite : a > 0 Et b > 0
-- Assure :
--      Résultat >= 1
--      a Mod Résultat = 0
--      b Mod Résultat = 0
--      « C'est le plus grand ! »
-- Exemples :
--  A < B : A = 10, B = 16 --> PGCD = 2
--  A = B : A = 10, B = 10 --> PGCD = 10
--  A > B : A = 21, B = 10 --> PGCD = 1
Fonction Pgcd (A, B : in Entier) retourne Entier
```

   La signature en C est :

```c
int pgcd(int a, int b);
```

5. Obtenir le quotient et le reste d'une division entière

Solution :

```
-- Le quotient et le reste d'une division entière
--
-- Paramètres :
--      Dividende: in Entier
--      Diviseur : in Entier
--      Quotient : out Entier
--      Reste : out Réel
--
-- Nécessite :
--      Diviseur >= 0
--      Diviseur > 0
--
-- Assure :
--      0 <= Reste
--      Reste < Diviseur
--      Dividende = Quotient * Diviseur + Reste
--
-- Exemples :
--      Nominal (reste non nul) : Dividende = 11, Diviseur = 4 -> Quotient = 2 et Reste = 3
--      Nominal (reste nul) : Dividende = 12, Diviseur = 4 -> Quotient = 3 et Reste = 0
Procédure Div_Mod (Dividende, Diviseur : in Entier ;
                   Quotient, Reste : out Entier)
```

La signature en C est :

```c
void div_mod(int dividende, int diviseur, int *quotient, int *reste);
```

Remarque : On pourrait aussi en faire une fonction si on pouvait renvoyer les deux valeurs (voir Enregistrement).

Remarque : On pourrait faire deux fonctions l'une appelée Quotient et l'autre Reste (mais on perd en efficacité car on fait deux fois les mêmes calculs).

Exercice 3 : Modéliser un robot de type 1

L'objectif de cet exercice est de modéliser un robot de type 1. Un tel robot se déplace dans un environnement qui peut être modélisé par un quadrillage. Chaque case du quadrillage correspond à une position possible du robot. Elle est repérée par son abscisse (x) et son ordonnée (y).

Le robot de type 1 est caractérisé par sa position (abscisse et ordonnée) dans l'environnement et sa direction. Seulement quatre directions sont possibles : haut, bas, droite ou gauche.

Un robot de type 1 peut avancer dans son environnement d'un pas (d'une case) suivant sa direction et il peut pivoter de 90^o sur sa droite.

1. Définir le type Robot1 qui définit un robot de type1. Il permettra par exemple de déclarer une variable représentant un robot positionné à l'abscisse 2 et ordonnée 5 et dans la direction « gauche ».

Solution :

```c
enum Direction { HAUT, DROITE, BAS, GAUCHE };
typedef enum Direction Direction;
```

```c
struct Robot1 {
    int x;        /* abscisse */
    int y;        /* ordonnée */
    Direction direction;
};

typedef struct Robot1 Robot1;
```

2. Écrire un sous-programme qui crée un robot de type 1 à partir de sa position initiale (abscisse et ordonnée) et de sa direction initiale. Par exemple, on peut créer un robot d'abscisse 2, ordonnée 5 et direction « gauche ».
Solution :

```c
Robot1 robot1(int x, int y, Direction d) {
    Robot1 robot = { x, y, d };
    return robot;
}
```

3. Écrire un sous-programme qui fait pivoter un robot de 90^o vers la droite.
Solution :

```c
void pivoter(Robot1 *robot) {
    robot->direction = (robot->direction + 1) % 4;
}
```

4. Écrire un sous-programme qui fait avancer un robot d'une seule case suivant sa direction courante.
Solution :

```c
void avancer(Robot1 *robot) {
    switch (robot->direction) {
        case HAUT:
            robot->y++;
            break;
        case BAS:
            robot->y--;
            break;
        case DROITE:
            robot->x++;
            break;
        case GAUCHE:
            robot->x--;
            break;
    }
}
```

5. Écrire un programme correspondant au scénario suivant.

1. Créer un robot r1 en position (4,10) orienté « droite ».
2. Créer un robot r2 en position (15,7) orienté « bas ».
3. Faire pivoter le robot r1.
4. Vérifier que la direction de r1 est « bas ».
5. Faire avancer le robot r2.
6. Vérifier que l'ordonnée de r2 est 6.

Solution :

```
void scenario_sujet() {
    Robot1 r1 = robot1(4, 10, DROITE);
    Robot1 r2 = robot1(15, 7, BAS);
    pivoter(&r1);
    assert(r1.direction == BAS);
    avancer(&r2);
    assert(r2.y == 6);
}
```

6. Expliquer comment faire un module robot1 à partir des éléments écrits ci-dessus.
Solution : Il s'agit de définir un fichier d'entête `robot1.h` qui correspond à la spécification (ou interface) du module et contient la définition des types et la spécification (signature et sémantique) des sous-programmes. On utilisera le préprocesseur en mettant les élements précédents entre :

```
#ifndef robot1__h
#define robot1__h

... définition des types et des spécifications des sous-programmes ...

#endif
```

On définit ensuite un deuxième fichier `robot1.c` qui contient l'implantation des sous-programmes spécifiés dans l'interface du module. S'il y avait des sous-programmes présents que dans l'implantation du module et pas dans l'interface, on les déclarerait `static`.