ISMIN 1A

Programmation 1 : Examen de programmation C **Réponses**

Partie 1. Questions ouvertes

1. Soit la déclaration suivante, `char tab[] = ""`. Que contient le tableau `tab` ?

   Réponse : `tab[0] = '\0'`.

   `tab` est une chaîne de caractères stockée dans un tableau d'un seul élément : `tab[0] = '\0'`.

2. Donner la déclaration d'un pointeur sur un tableau de 10 chaînes de caractères.

   Réponse : Deux possibilités, selon l'utilisation des « chaînes de caractères » :

   (a) Un tableau de caractères : `char (*pointeur)[10][LONG];`

   (b) Un pointeur : `char *(*pointeur)[10];`

3. (*2) Donner la déclaration d'un tableau dont chacun de ses 20 éléments est une structure à trois champs : un entier `nombre`, un pointeur d'entier `ptrint` et un tableau de 15 caractères. Supposant ce tableau initialisé, donnez l'expression permettant de placer dans le champ `ptrint` du 4ème élément du tableau, l'adresse du champ `nombre` du 11ème élément.

   Réponse : `tableau[3].ptrint = &tableau[10].nombre`

   Avec les déclarations suivantes :

   ```
   typedef struct examen_t {
        int nombre;
        int *ptrint;
        char chaine [15];
   } Examen;

   Examen tableau[20];
   ```

4. Soit la déclaration suivante : `char les[] = "ab"`.
   Que vaut `les[2]` ?

   Réponse : `les[2]` vaut `'\0'`.

   Le tableau `les` est une chaîne de caractères, car il a été initialisé avec une chaîne (`"ab"`). Le nombre d'éléments de ce tableau est donc le nombre de caractères présent dans la chaîne, plus 1, le caractère de fin de chaîne. En conséquence, `les[2]` contient celui-ci, c'est à dire `'\0'`.

5. Donnez une forme équivalente à l'expression `&tab[0]`.

   Réponse : `&tab[0]` ≡ `tab`.

6. Soit la déclaration suivante : `short ly[5] = {1, 2 }`.
   Que vaut `ly[3]` ?

   Réponse : `ly[3]` vaut `'\0'`.

   La taille du tableau (5) est supérieure au nombre d'éléments indiqués dans l'initialisation (2). Dans ce cas, les éléments manquants sont initialisés à zéro, c'est à dire `'\0'`.

7. Donnez une forme équivalente à l'expression `p->champ`.

   Réponse : `p->champ` ≡ `(*p).champ`

   `p` est un pointeur vers une structure dont un champ a pour nom `champ`.

   Les parenthèses sont nécessaires, car l'opérateur `.` est de priorité plus élevée que l'opérateur unaire `*`.

8. Dans le code suivant :

```
char ch[] = "Bonjour", *pc = ch;
while (*pc) pc++;
```

Que vaut `pc` après l'exécution de l'instruction `while` ? Vers quel caractère pointe-t-il ?

Réponse : `pc` vaut `&pc[7]` et le caractère pointé est le caractère de fin de chaine `'\0'`.

L'instruction `while` peut s'écrire ainsi :

```
while ( *pc != '\0' ) {
     pc = pc + 1;
}
```

Comme `pc` est initialisé avec l'adresse du premier caractère de la chaine `ch`, la boucle va être exécutée tant que le caractère pointé n'est pas le caractère de fin de chaine. À la sortie de la boucle, `pc` vaut `&pc[7]` et le caractère pointé est le caractère de fin de chaine `'\0'`.

9. (*2) Soit le code suivant :

```
void g(int *p) {
     *p = 12;
}

int main() {
     int *p;
     g(p);
     printf("%d\n", *p);
}
```

Qu'est ce qui s'affiche ?

Réponse : L'affichage est indéterminé ou erreur d'exécution.

Le pointeur `p` défini dans la fonction `main` n'est pas initialisé. Comme il s'agit d'une variable locale, son contenu est indéterminé.

L'expression `g(p)`, puis, dans la fonction `g`, p`p = 12`, sont donc erronées. Avec de la chance, il y aura une erreur à l'exécution (*segmentation error*), ou pire, un écrasement mémoire.

10. Que vaut la variable `j` à la fin de la fonction `f` ?

```
void f(int i) {
     int *j;
     *j = i;
}
```

Réponse : Son contenu est indéterminé.

`j` est un pointeur, qui n'est pas initialisé. Comme il s'agit d'une variable locale, son contenu est indéterminé.

L'expression `*j` est donc erronée. Avec de la chance, il y aura une erreur à l'exécution (*segmentation error*), ou pire, un écrasement mémoire.

11. (*2) Que vaut la variable `fleuri` dans le code suivant :

```
char fleuri = 100;
fleuri *= 2;
```

Pourquoi ?

Réponse : `fleuri` vaut `-56`.

Le type `char` est signé. Sa plage de valeurs est de `-128` à `127` (de 2^{n-1} à $2^{n-1}-1$), où n est le nombre de bits du type, ici 8. Concernant les nombres signés, pour un nombre a positif donné, nous avons la formule suivante : $a - 2^n = (-a)$. Ici, pour `fleuri` égal à `200`, le nombre négatif correspondant est donc `-56`.

12. Soit `ge`, un tableau de 10 caractères. Donnez l'instruction permettant de saisir au clavier une chaîne de caractères, sans utiliser le caractère `'&'`.

    Réponse : `scanf("%s", ge);`

13. Soit `erne`, un tableau de 10 entiers. Donnez l'instruction permettant de saisir au clavier le 3e entier.

    Réponse : Soit

    ```
    scanf("%d", &erne[2]); ou bien
    scanf("%d", erne + 2);
    ```

14. (*2) Pourquoi doit-on placer un `'&'` devant la variable `ime` dans le code suivant :

    ```
    int ime;
    scanf("%d", &ime);
    ```

    Réponse : La fonction `scanf` permet de lire des octets à partir du fichier `stdin`, habituellement relié au clavier, de les convertir éventuellement selon le format indiqué (ici `%d`, pour une conversion en entier, de caractères numériques en base 10) et de stocker le résultat de cette conversion dans une variable.

    En langage C, « stocker dans une variable » s'exprime en utilisant l'opérateur `&`.

15. Que vaut la variable `ernet` à la fin du code suivant ?
    On rappelle que `x += y` est équivalent à `x = x + Y` et que de manière générale `x` **`op`** `= y` est équivalent à `x = x` **`op`** `y`.

    ```
    int ernet = 100;
    ernet += 5;
    ernet *= 10;
    ernet /= 15;
    ernet %= 4;
    ```

    Réponse : `ernet` vaut 2.

16. (*3) Soit le code suivant :

    ```
    int i;
    for (i=0; i<10; i++) {
         printf("i = %d\n", i);
    }
    ```

    Réécrivez-le en utilisant une instruction `while`.

    Réponse :

    ```
    int i = 0;
    while (i<10) {
         printf("i = %d\n", i);
         i++; }
    ```

17. Soit la déclaration suivante, `int *tab[10]`. Indiquez ce qu'est `tab` ?

    Réponse : `tab` est un tableau de 10 pointeurs d'entiers.

18. Dans le code suivant :

    ```
    int i=0, j=0;
    j = i++;
    ```

    Que vaut la variable `j` après l'instruction d'affectation ?

    Réponse : `j` vaudra 0.

    L'expresssion `i++` se lit de gauche à droite : d'abord, le retour de sa valeur, qui sera stockée dans `j`, puis l'incrémentation de `i`. Ainsi, `j` vaudra 0.

19. (*2) Donnez l'expression C permettant de calculer $i = i * 2^n$.

Réponse : L'expression est `i «= n`.

Multiplier par 2 consiste « à ajouter un zéro à droite » (poids faible), donc décaler la valeur de 1 vers la gauche. Multiplier par 2^n, consiste ainsi à décaler la valeur de n vers la gauche. En C, l'opérateur de décalage vers la gauche est «.

L'expression est donc : `i «= n`.

Note. Les calculs effectués avec une boucle ont été considérés comme corrects.

20. (*3) Soit le code suivant :

```
char tab[] = "bonjour";
int i;
for (i=0; tab[i] != '\0'; i++) {
     tab[i] = tab[i] - 'a' + 'A';
}
```

Que fait ce programme ?

Proposez une version sans utiliser l'opérateur d'accès tableau (`[]`).

Réponse : Ce programme transforme la chaine de caractères `"bonjour"` en majuscules.

```
char tab[] = "bonjour", *pc;
for (pc = tab; *pc != '\0'; pc++) {
     *pc = *pc - 'a' + 'A';
}
```

21. (*3) La fonction `strcpy` permet de copier une chaine de caractères dans une autre, existante. Voici son prototype :

`char *strcpy ( char *destination, char *source);`.

Écrivez cette fonction **sans** utiliser de tableaux ni d'opérateurs `[]` (uniquement des pointeurs et des accès pointeurs).

Réponse :

```
char *strcpy ( char *destination, char *source) {
     while (*source != '\0') {
          *destination = *source;
          *destination++;
          *source++;
     }
     *destination = '\0';
}
```

22. Soit la déclaration suivante : `char mant[] = { 'a', 'b' }`.

Que vaut `mant[2]` ?

Réponse : Son contenu est indéterminé.

Le tableau `mant` est implicitement déclaré de taille le nombre d'éléments présent dans son initialisation, donc ici 2. Il n'y a donc pas d'élément `mant[2]`.

`mant` n'est pas une chaîne de caractères, car il n'a pas été initialisé avec une chaîne (`"ab"`).

23. Dans le code suivant :

```
int i=0, j=0;
j = ++i;
```

Que vaut la variable `j` après l'instruction d'affectation ?

Réponse : `j` vaudra 1.

L'expresssion `++i` se lit de gauche à droite : d'abord l'incrémentation de `i`, puis retour de sa valeur, pour être stockée dans `j`. Ainsi, `j` vaudra 1.

24. Que vaut la variable `ille` dans le code suivant ?

```
float ille;
ille = 2/3*100.0;
```

Pourquoi ?

Réponse : `ille` vaut `0.0`.

L'expression `2/3` est évaluée dans le type `int`, donc vaut 0.

`Note.` En cas d'opérateurs de même priorité, la norme du langage C ne précise pas l'ordre d'évaluation de l'expression. Pour éviter tout problème, il faut bien indiquer les types des constantes et placer des parenthèses pour éviter l'ambiguité : `2.0/(3.0*100.0)` ou `(2.0/3.0)*100.0`.

25. (*2) Soit le code suivant :

```
int i=0, tab[10], n=27;
do {
     tab[i++] = n % 10;
} while ((n /= 10) > 10);
```

Que contient le tableau `tab` en fin de programme ?

Réponse : `tab[0]` vaut 7 et les autres cases du tableau ont une valeur indéterminée.

Comme il s'agit d'une boucle `do while`, l'instruction associée est exécutée au moins une fois. L'expression `i++` se lit de gauche à droite, c'est à dire retourne la valeur de `i` avant de l'incrémenter, soit 0. Donc `tab[0]` vaut le reste de la division entière entre 27 (`n`) et 10, soit 7. `tab[0]` vaut 7.

Le test de l'instruction est faux car `n / 10` vaut 2 qui est inférieur à 10. Les autres cases du tableau ont donc une valeur indéterminée.

26. (*3) Soit le code suivant :

```
int i;
printf("i ? ");
scanf("%d", &i);
switch (i) {
     case 0 : printf(" NUL"); break;
     case 1 : case 3 : case 5 : case 7 : case 9 :
          printf(" IMPAIR"); break;
     case 2 : case 4 : case 6 : case 8 :
          printf(" PAIR"); break;
     default : printf("NEGATIF OU PAS UN CHIFFRE"); break;
}
```

Réécrivez l'instruction `switch` en n'utilisant que des instructions `if`.

Réponse :

```
int i;
printf("i ? ");
scanf("%d", &i);
if (i == 0) {
     printf(" NUL");
}
else if (i == 1 || i ==3 || i == 5 || i == 7 || i == 9) {
     printf(" IMPAIR");
}
else if (i == 2 || i ==4 || i == 6 || i == 8) {
     printf(" PAIR");
}
else {
     printf("NEGATIF OU PAS UN CHIFFRE");
}
```

27. Donnez la déclaration d'une fonction `g` sans paramètre et qui retourne l'adresse d'un entier.

Réponse : `int *g(void);`

28. Donnez la déclaration d'une fonction `f` sans paramètre et qui ne retourne aucune valeur.

Réponse : `void f(void);`

29. (*2) Soit le code suivant :

```
int i = 0, j = 5, somme = 20;
if (j && (i = somme / j) ) somme = 10;
else somme = 30;
```

Donnez les valeurs des variables après l'exécution de ce programme.

Réponse : `i` vaut 4, `j` vaut 0 et `somme` vaut 10.

Le test dans l'instruction `if` peut s'écrire ainsi :

```
j != 0 && (i = somme / j) != 0
```

Comme `j` est différent de zéro (il vaut 5), le premier terme est vrai. Le résultat de l'expression `somme/j` est d'abord calculé, donc la valeur 4 est stockée dans `i`, qui différente de 0, entraîne que le second terme est vrai également.

En conséquence, `i` vaut 4, `j` vaut 0 et `somme` vaut 10.

30. (*2) Soit le code suivant :

```
#include <stdio.h>
#define MAX = 10

int main() {
     int tab[MAX];
     tab[2] = 2;
     printf("%d", tab[MAX-8]);
}
```

Ce programme compile-t-il ? Si oui qu'affiche-t-il ?

Réponse : Ce programme ne compile pas.

La ligne `#define MAX = 10` doit s'écrire `#define MAX 10` (sans le signe =).