Exercices programmation – Révision C – v3

Les exercices ci-dessous sont principalement basés sur les erreurs faites lors des tests et projets des années précédentes. Vous êtes encouragé à tester les codes ci-dessous sur votre machine. Si vous avez des doutes ou des questions n'hésitez pas à indiquer le numéro de la question sur le forum Ed.

Q1. Que retourne `F1()`? Pourquoi ? idem si k est défini comme un `short`? et `char`?
Est-ce possible de faire `return k` au lieu de `a` ? Pourquoi ?

```
int F1(void){
    int a=0;
    for (int k=10;k>0;k++) {a=k;}
    return a;
}
```

F1() retourne 2147483647 soit 2^{31} -1. En effet k est initialisée dans la « for loop » comme étant un int. Les ints peuvent prendre comme valeur tout entier compris entre -2^{31} et 2^{31} -1 car ils prennent 4 bytes de mémoire soit 4*8=32 bits.

Si k est défini comme un short, alors F1() retourne 32767 soit 2^{15} -1. En effet, les shorts peuvent prendre comme valeur tout entier compris entre -2^{15} et 2^{15} -1 car ils prennent 2 bytes de mémoire soit 2*8=16 bits.

Si k est définie comme un char, alors F1() retourne 127 soit 2^{7} -1. En effet, les chars peuvent prendre comme valeur tout entier compris entre -2^{7} et 2^{7} -1 car ils prennent 1 bytes de mémoire soit 1*8=8 bits.

Nous ne pouvons pas faire return k au lieu de a, car k est seulement connu dans la « for loop » i.e. k est une variable local à la for loop uniquement.

Q2. Est-ce que `F2()` compile ? Pourquoi ? Quelle est la taille de A[] ?

```
void F2(void){
    char A[];
}
```

F2() ne compile pas car nous n'attribuons pas de taille à A (i.e. il manque un chiffre entre les […]) lors de sa déclaration OU nous ne l'initialisons pas directement (ex : char A[]= « hello »). A n'a pas de taille.

Q3. Que se passe-t-il lors de l'exécution de `F3()`? Pourquoi ?

```
void F3(void){
    char *B;
    B[1] = 'x';
}
```

Crash lors l'exécution de F3() et une erreur/warning lors de la compilation. En effet, nous essayons d'affecter une valeur à un pointeur qui ne pointe pas vers une autre valeur (i.e. un pointeur qui n'a pas été initialisé). De plus l'espace pour B[1] n'est pas réservé.

Q4. Que se passe-t-il lors de l'exécution de `F4()`? Pourquoi ?

```
void F4(void){
    char* D="hello";
    D[0] = 'y';
}
```

Crash lors l'exécution F4() et une erreur/warning lors de la compilation. En effet, nous initialisons le pointeur D vers la chaîne de caractères « hello » met D est en lecture seuls. il y aura un crash si nous essayons décrire à cet endroit.

Q5. Que se passe-t-il lors de l'exécution de `F5()`? Pourquoi ?

```
void F5(void){
    char D[]="hello";
    D[0] = 'y';
}
```

Dans un premier temps, le tableau de char D est initialisé avec pour valeurs :
D[0]= 'h'
D[1]= 'e'
D[2]= 'l'
D[3]= 'l'
D[4]= 'o'
D[5]= \0

Ensuite la valeur de D[0] est changée pour 'y'. Nous obtenons donc :
D[0]= 'y'
D[1]= 'e'
D[2]= 'l'
D[3]= 'l'
D[4]= 'o'
D[5]= \0

Q6. Qu'affiche `F6()`? Pourquoi ?

```
void F6(void){
    char D[]="hello";
    D[5] = 'y';
    printf("%s\n",D);
}
```

F6() affiche helloy.
En effet, dans un premier temps, le tableau de char D est initialisé avec pour valeurs :
D[0]= 'h'
D[1]= 'e'
D[2]= 'l'
D[3]= 'l'
D[4]= 'o'
D[5]= \0

Puis le char de fin de ligne \0 est remplacé par 'y' , D[5]= 'y'

Nous obtenons donc :
D[0]= 'h'
D[1]= 'e'
D[2]= 'l'
D[3]= 'l'
D[4]= 'o'
D[5]= 'y'
printf() imprime les caractères de D jusqu'à ce qu'un caractère \0 soit rencontré,
-> affichage : helloy##..# avec ##..# correspondant à 0..n caractères quelconques

Q7. Que vaut F? Pourquoi ?

```
int E[]= {4,5,6};
int F = sizeof(E)/sizeof(E[0]);
```

F vaut 3. En effet sachant que E[] est un tableau de int et que nous savons que les int prennent 4 bytes de mémoire, le tableau prend 12 bytes de mémoire (3 éléments) (i.e. sizeof(E)=12) et chaque élément du tableau prend individuellement 4 bytes de mémoire (i.e. sizeof(E[0])=4). Ainsi 12/4=3 !

Q8. Que vaut e après l'appel à F8()? Pourquoi ? idem si E[]= {7}; ?

```
int F8(int E[]) {
        return sizeof(E)/sizeof(E[0]);
}

int E[]= {4,5,6};
int e = F8(E);
```

e vaut 2 après l'appel à F8(). En effet, sizeof(E) renvoie ici la taille d'un pointeur int* soit 8 bytes (pour architecture 64 bits) alors que sizeof(E[0]) renvoie la taille d'un int soit 4 bytes. Ainsi 8/4=2 !

Q9. Qu'affiche F9()? Pourquoi ? Quel est la taille de A[] et B[] ?

```
void F9(void){
    char A[]={'x'};
    char B[]="x";
    printf("%s, %s\n",A,B);
}
```

F9() affiche x##..#, x avec ##..# correspondant à 0..n caractères quelconques
La taille de A est de 1 char, il n'y a pas le \0 de fin de ligne. B correspond à 2 char 'x' et \0.
Ainsi lorsque nous utilisons printf pour afficher A n'est pas défini du fait qu'il manque le char \0, les char après A[0] seront affichés jusqu'à ce qu'un \0 soit rencontré.

Q10. Qu'affiche F10()? Pourquoi ?
Si vous êtes sur OSX/linux essayer votre code sur Windows et vice-versa, y a-t-il une différence, pourquoi ?

```
void F10(void){
    int i = 7;
```

```
    printf("%d %d\n", i, i++);
}
```

Le compilateur devrait vous indiquer qu'il y a un problème « unsequenced modification to 'i'" en effet la norme ne définit pas l'ordre d'exécution entre 2 *sequence points*. Windows : 8 7, OSX : 7 7

Q11. Est-ce que `F11()` compile ? Pourquoi ?

```
void F11(void){
    int* i, j;
    *j=3;
}
```

F11() ne compile pas, car j est un int et pas un pointeur sur un int, j contient une valeur et non une adresse !

Q12. Est-ce que `F12()` compile ? Pourquoi ?
Est-ce que `F12()` s'exécute correctement ? Pourquoi ?

```
void F12(void){
    char *p;
    *p = malloc(10);
}
```

F12() compile mais avec un « Warning »:

warning : assignment to 'char' from 'void *' makes integer from pointer without a cast

Et F12() crash lors de l'exécution car malloc() retourne un pointeur et on essaye de l'assicher à un char.
Le format correct est :

p = (char *)malloc(10) ;

Q13. Est-ce que `F13()` compile ? Pourquoi ? Est-ce que `F13()` s'exécute correctement ?
Pourquoi ? Essayez votre code sur un autre OS, avez-vous le même résultat ?

```
void F13(void){
    double a[1024][1024];
    a[0][0] = 0;
}
```

Il est possible que F13() compile, mais il va surement crasher lors de l'exécution. Il faut garder en tête que cela prend beaucoup de place : 1024*1024*8=8388608 bytes soit 8 MB et l'OS pourrait ne pas pouvoir réserver autant de place sur la stack.

Q14. La/lesquelles de ces déclarations de `main()` est/sont correcte(s)? Pourquoi ?

```
int main(void)
```

```
int main(int argc, const char * argv[])
int main(int argc, const char * argv[], const char *envp[])
```

Elles sont toutes correctes.
En effet, int main(void) indique que le programme ne peut pas être appelé avec un argument du fait du void.

int main(int argc, const char * argv[]) indique que le programme peut être appelé avec des arguments, avec argc le nombre d'arguments et *argv un tableau de pointeurs qui pointent vers ces différents arguments. Attention ici les pointeurs peuvent changer de valeur mais pas les arguments eux-mêmes du fait du const.

int main(int argc, const char * argv[], const char *envp[]) indique que le programme peut être appelé, avec argc le nombre d'arguments et *argv un tableau de pointeurs qui pointent vers ces différents arguments. Attention ici les pointeurs peuvent changer de valeur mais pas les arguments eux-mêmes du fait du const. Ici nous nous intéressons aussi à l'environnement : en effet *envp est un tableau de pointeur qui pointent vers les variables environnementales. Attention ici les pointeurs peuvent changer de valeur mais pas les variables environnementales elles-mêmes du fait du const. Ce 3e paramètre n'est pas défini pour tout les OS.

Q15. Quel est la valeur minimale de `argc` ? que contient `argv[0]` ?

```
int main(int argc, const char * argv[])
```

La valeur minimale de argc est 1. Argv[0] contient le premier argument de la ligne de commande soit ./le_nom_du_programme.

Q16. L'exécutable `myProg` est appelé avec `./myProg 42`
Que valent `argc, argv[0], argv[1], argv[2]` ?
Comment afficher le paramètre 42 ?

```
int main(int argc, const char * argv[]){

        printf("%...", argv[...]) ;
}
```

argc=2
argv[0]= "./myProg"
argv[1]= "42"
argv[2]= NULL
Pour afficher le paramètre 42, nous complétons avec %s et argv[1].

Q17. Comment convertir un `char` contenant un **chiffre** en un `int`?

```
char c='8';
int  cc = … ; // cc doit valoir 8
```

int cc=atoi(&c) ; ou int cc=c-'0' ;

Q18. Qu'affiche `F18()`? Pourquoi ? Essayez votre code sur un autre OS, avez-vous le même résultat ? Comment réécrire le code ci-dessous pour enlever l'ambiguïté ?

```
void F18(void){
    int i = 3;
    i = i++;
    printf("i: %d\n",i);
}
```

Le compilateur devrait vous indiquer qu'il y a un problème « unsequenced modification to 'i'" en effet la norme ne définit pas l'ordre d'exécution entre 2 *sequence points*. OSX : 3, Windows : 4.
Pour éviter toute confusion et si nous voulons incrémenter i de 1 nous pouvons utiliser i++ ou i=i+1 ou i+=1 ;

Q19. Qu'affiche `F19()`? Pourquoi ? Essayez votre code sur un autre OS

```
int aa(void){printf("aa "); return 1;}
int bb(void){printf("bb "); return 2;}
void F19(void){
    printf("%d %d\n", aa(),bb());
}
```

Le compilateur devrait vous indiquer qu'il y a un problème « unsequenced operations » F19() peut afficher bb aa 1 2 ou alors aa bb 1 2. Il est important de noter que l'ordre de lecture/d'évaluation des arguments dans printf n'est pas spécifié, donc il peut varier d'une machine à une autre. Ceci veut donc dire que aa et bb peuvent être imprimés dans n'importe quel ordre, mais les valeurs 1 et 2 seront toujours affichées dans cet ordre.

Q20. `F20()` regarde si b se trouve entre a et c, pourquoi le résultat est incorrect?

```
int F20(int a,b,c) {
    if (a < b < c) return  1;
              else return -1;
}
```

Le résultat est incorrect car la règle de priorité associée à l'opérateur « < » force le programme à faire la comparaison suivante (a<b)<c où a<b renvoie 0 si a>b et 1 si a<b.

Q21. Qu'affiche `F21(1000)`? Pourquoi ?

```
void F21(short a){
    short b= a*a;
    printf("%d x %d = %d\n", a,a,b);
}
```

F21(1000) affiche 16960 car 1000 * 1000 % 2^16= 1 000 000 % 65536 = 16960 (RAPPEL : les shorts prennent 2 bytes de mémoire soit 2*8=16 bits).

Q22. Qu'affiche `F22(3)`? Pourquoi ?

```
void F22(int a){
    double b= a/2;
    printf("%d/2 = %f\n", a,b);
}
```

F22(3) affiche 1 car en C, si les variables sont du même type (des int dans notre cas), il n'y a pas de conversion préalable et le résultat est un int peu importe l'opération utilisée et le type de variable à laquelle nous attribuons le résultat de la division.

Q23. Qu'affiche `F23(3)`? Pourquoi ?

```
void F23(int a){
    double b= a/2.0;
    printf("%d/2.0 = %f\n", a,b);
}
```

F23(3) affiche 1.500000 car dans ce cas le diviseur est un double, le compilateur va donc préalablement faire une conversion pour le type du diviseur, dans notre cas de int à float/double. La division sera donc double/double -> double.

Q24. Qu'affiche `F24(3)`? Pourquoi ?

```
void F24(double a){
    int b= a/2;
    printf("%f/2.0 = %d\n", a,b);
}
```

F24(3) affiche 1 car dans ce cas bien que le dividende soit un double qui entraine une conversion du diviseur et donc une division double/double, le resultat est stocker dans un int -> valeur double convertie/tronquée en int. 3.0/2 = 1.5 -> 1

Q26. Qu'affiche `F25()` ? Pourquoi ?

```
void F25(void){
    double a=12.34;
    int    b=456;
    printf("1) a:%f, b: %d\n", a,b);
    printf("2) a:%d, b: %f\n", a,b);
    printf("3) a:%d, b: %d\n", a,b);
    printf("4) a:%f, b: %f\n", a,b);
    printf("5) a:%d, b: %d\n", (int)a,b);
    printf("6) a:%d, b: %d\n", (double)a,(double)b);
    printf("7) a:%f, b: %f\n", (double)a,(double)b);
}
```

F25() affiche :
1) a:12.340000, b: 456
2) a:456, b: 12.340000
3) a:456, b: 456 ou a:456, b: 0 (fonction de l'OS)
4) a:12.340000, b: 0.000000
5) a:12, b: 456
6) a :-234634592, b : 0 ou a:2, b: 0 (fonction de l'OS)
7) a :12.340000, b : 456.000000

Cet affichage se justifie par deux choses :
- Les règles de lecture/d'évaluation des arguments dans printf en fonction du format string utilisé.
- Utiliser printf avec le mauvais format string est un comportement indéfini (« undefined behaviour »). En effet cela vient du fait que les doubles et les ints sont de différentes tailles (64 bits pour les doubles, 32 bits pour les ints) !

Q26. Est-ce `F26(100)` s'exécute correctement sur votre machine?
Et pour `F26(10000000)`? Pourquoi ?
Comment modifier `F26()`pour qu'elle s'exécute pour toutes les valeurs de sz ?

```
void F26(unsigned int sz){
    int a[sz];
    a[0]=2;
}
```

F26(100) s'exécute correctement. Toutefois cela n'est pas le cas pour F26(10000000) qui va crasher car il n'y aura pas assez de place sur la stack pour mettre a[].

Q27. Quelle est la différence entre les 2 lignes ci-dessous :

```
const int MAXSIZE = 100;
#define MAXSIZE 100
```

Les #defines sont pris en charge par le pre-processeur et lors de la compilation les #defines sont remplacés par la chaine de caractère après le define, c'est purement « visuel » alors que consts int est une variable avec un espace mémoire associé, le const indique que cette variable est en lecture seule.

Q28. Qu'affiche `F28()`? Pourquoi ?

```
void F28(void){
    char a[5]="hello";
    printf("%s\n",a);
}
```

F28() affichera probablement hello bien que a[5] ne contienne pas assez de place pour contenir le caractère de fin de string- > a[6] pour contenir \0.
Il pourrait aussi afficher hello##..# avec ##..# correspondant à 0..n caractères quelconques

Q29. Qu'affiche `F29()`? Pourquoi ?

```
void F29(void){
    printf("%.1f, %.20f\n",3.1, 3.1);
}
```

F29() affiche 3.1, 3.10000000000000008882. En effet, 3.1 ne peut pas être exactement représenté précisément par un float. Ainsi ce qui sera imprimé lors de l'appel de F29 sera dans un premier temps 3.1 avec un chiffre décimal de précision (%.1f) soit 3.1 puis la valeur la plus proche de 3.1 avec 20 chiffres décimaux de précision (%.20f) que votre machine peut représenter soit ici 3.10000000000000008882.

Q30. Combien de fois s'exécute la boucle dans `F30()`? Pourquoi ?

```
void F30(void){
    for (double x=0.0; x != 10.0; x += 0.1)
       printf("x:%f\n",x);
}
```

La boucle de F30() est une boucle infinie car dans ce cas dû à la représentation des doubles x ne sera jamais exactement égale à 10, mais 9.99999999999999999984 !
Il faut réécrire le test comme suit : for (double x=0.0; x <= 10.0; x += 0.1)

Q31. Qu'affiche `F31(5)`? Pourquoi ?

```
int F31(int a){
    if (a>2)
        printf("a>2\n");
    if (a>5) ;
        printf("a>5\n");
    return a;
    printf("a: %d\n",a);

}
```

F31() affiche
a>2
a>5
En effet, nous constatons dans un premier temps 5>2 donc le premier printf est appelé. En prêtant bien attention au code nous voyons qu'il y a un point-virgule après le deuxième if ce qui implique que quand vient même a>5 le programme ne fait rien de plus que si a ≤5. Ensuite le deuxième printf est appelé et enfin le code renvoie a et se termine, sans appeler le dernier printf car ce dernier se trouve après la ligne du return !

Q32. Sur ma machine `F32()` affiche '8', mais sur la VM elle affiche '4' ? Pourquoi ?

```
void F32(void){
    int *p;
    printf("sizeof(p): %ld\n", sizeof(p));
}
```

Ceci est dû à une différence d'architecture entre les deux machines. En effet les machines qui ont une architecture 32 bits attribuent une taille de 4 bytes aux pointeurs alors que les machines qui ont une architecture 64 bits attribuent une taille de 8 bytes aux pointeurs.