

Examen de Langage C

Durée : 1h30

Aucun document autorisé – Mobiles interdits

Notez que les affirmations (antécédents, conséquents, rôles, et invariants) dans vos codes C entreront pour partie dans la note finale.

- 1. En C, combien existe-t-il de modes de transmission des paramètres ?

En C, il n'existe d'un seul mode de transmission des paramètres : la transmission par valeur.

- 2. Expliquez ce ou ces modes de transmission des paramètres de la question 1.

Lors d'une transmission par valeur au moment de l'appel d'une fonction, le paramètre effectif est évalué. La valeur calculée, résultat de l'évaluation, est ensuite affectée au paramètre formel correspondant. Ce mode de transmission est adapté à la transmission des données. Il résulte de ce mécanisme qu'après l'appel d'une fonction, le paramètre effectif conserve sa valeur d'avant l'appel, mais si le paramètre formel correspondant est modifié dans la fonction.

- 3. Soit la procédure C : `void p(int x) {x++;}`. Soit le code : `int y=1; p(y);`. Quelle est la valeur de y après l'appel de p ? Expliquez.

Après l'appel de p, y reste égal à 1 car le mode de transmission en C est uniquement par valeur, adaptée aux paramètres donnée et pas aux paramètres résultat.

- 4. Soit le fragment de code algorithmique suivant :

```
si x=1 alors y ← a finsi
si x=2 alors y ← b finsi
si x≠1 et x≠2 alors y ← c finsi
```

Expliquez pourquoi ce code est mal écrit, et récrivez-le correctement.

ce fragment de code est mal écrit car il fait des tests sur des conditions qui s'excluent mutuellement. Ainsi, par exemple, si x=1, le code testera inutilement x=2. La bonne écriture doit utiliser la partie sinon de l'énoncé conditionnel si.

```
si x=1 alors y ← a
```

```
sinon { x≠1 }
  si x=2 alors y ← b
  sinon { x≠1 et x≠2 }
    y ← c
  finsi
finisi
```

- 5. Dans l'affectation : `{ A } v ← e { C }`, qui affecte le résultat de l'évaluation de l'expression e à la variable v, comment détermine-t-on le conséquent {C} à partir de l'antécédent {A} ?

Le conséquent {C} est obtenu en remplaçant toutes les apparitions de e par v dans l'antécédent {A}.

- 6. Soit la suite d'affectations avec l'antécédent initial :

```
{ y = x^2, d = 2x - 1 }
d ← d+2
y ← y+d
d ← d+2
y ← y+d
{ ??? }
```

Donnez chaque conséquent qui suit chacune des affectations. Quel est le résultat obtenu, si on répète i fois les 2 affectations `d ← d+2` et `y ← y+d` ?

```
{ y = x^2, d = 2x - 1 }
d ← d+2
{ y = x^2, d = 2x + 1 }
y ← y+d
{ y = (x+1)^2, d = 2x + 1 }
d ← d+2
{ y = (x+1)^2, d = 2x + 3 }
y ← y+d
{ y = (x+1)^2 + 2x + 3, d = 2x + 3 }
y ← y+d
{ y = (x+2)^2, d = 2x + 3 }
```

On remarque que cette suite d'affectations permet le calcul de $(x+2)^2$. Si on applique i fois cette suite, on calcule $(x+i)^2$.

- 7. Écrivez la fonction produit qui calcule le produit des chiffres d'un entier (int) passé en paramètre. Par exemple, `produit(1345)=60` et `produit(10415)=0`.

```
/*
```

```

/* Rôle : renvoie le produit des chiffres qui composent l'entier n */
int produit(int n) {
    if (n==0) return 0;
    //
    if (n<0) n=-n;
    // n>0 ⇒ décomposer n en base 10
    int prod=1;
    do {
        prod*=n%10;
        n/=10;
    } while (n!=0);
    //
    return prod;
}

```

- 8. Deux entiers naturels sont des nombres amiables si chacun est égal à la somme des diviseurs de l'autre. Par exemple, 220 et 284 sont des nombres amiables. Écrivez la fonction `êtesVousAmiables` qui teste si deux nombres sont amiables ou pas. La fonction renvoie un booléen.

```

/*
 * Antécédent : x ≥ 0
 * Rôle : renvoie la somme des diviseurs de x
 */
int sommeDiv(const int x) {
    int somme=1, divMax=x/2;
    // chercher et sommer tous les diviseurs de x sur [2;divMax[
    for (int i=2; i<=divMax; i++)
        if ((x%i)==0) somme+=i;
    //
    return somme;
}

/*
 * Antécédent : a ≥ 0 et b ≥ 0
 * Rôle : renvoie vrai si a et b sont des nombres amiables, c-à-d
 * si chacun est égal à la somme des diviseurs de l'autre
 */
int êtesVousAmiables(const int a, const int b) {
    assert(a>=0 && b>=0);
    return sommeDiv(a)==b && a==sommeDiv(b);
}

```

- 9. La déclaration de type `enum grade {A, B, C, D, E}`; représente le grade que peut obtenir un étudiant avec une note (comprise entre 0 et 100) obtenue à un examen. Écrivez la fonction `QuelGrade` qui possède un paramètre `n` de type `double` qui représente une note et qui renvoie comme résultat le grade qui correspond à cette note.

grade A : $n \geq 90$
 grade B : $80 \leq n < 90$
 grade C : $70 \leq n < 80$

grade D : $60 \leq n < 70$
 grade E : $n < 60$

```

/* Antécédent : n un réel double tel que 0 ≤ n ≤ 100 */
/* Rôle : renvoie le grade correspondant à la note n */
enum grade QuelGrade(const double n) {
    assert(n>=0 && n<=100);
    if (n>=90) return A;
    // sinon 0 ≤ n < 90
    if (n>=80) return B;
    // sinon 0 ≤ n < 80
    if (n>=70) return C;
    // sinon 0 ≤ n < 70
    if (n>=60) return D;
    // sinon 0 ≤ n < 60
    return E;
}

```

- 10. Écrivez la fonction `bonsGrades` qui renvoie le nombre de grades égaux à A ou B contenus dans un tableau `t` de `n` grades. Le tableau `t` et son nombre d'éléments `n` sont passés en paramètres de la fonction.

```

/*
 * Antécédent : t tableau initialisé de n grades
 * Conséquent : renvoie le nombre de grades A ou B dans t
 */
int bonsGrades(const int n, const enum grade t[]) {
    int nbBonsGrades=0;
    for (int i=0; i<n; i++)
        if (t[i]==A || t[i]==B)
            nbBonsGrades++;
    //
    return nbBonsGrades;
}

```