Autotest de maîtrise du langage C

Chaque question propose plusieurs réponses possibles dont une, plusieurs ou éventuellement zéro sont correctes. Il faut cocher toutes les réponses correctes.

▶ **Q. 1** Le langage C est un langage
☐ orienté objet ☐ fonctionnel ☑ impératif ☑ typé statiquement ☐ typé dynamiquement

▶ **Q. 2** Le langage C fut développé par
☑ D. Ritchie et K. Thompson ☐ B. Kernighan et D. Ritchie
☐ B. Stroustrup ☐ B. Gates ☐ S. Jobs

▶ **Q. 3** L'instruction `n = n + 1;` est équivalente à
☑ `n++;` ☑ `++n;` ☐ `*n++;` ☐ `*++n;` ☑ `n += 1;` ☐ `n + 1 = n;`

▶ **Q. 4** La déclaration d'un pointeur `p` sur un entier se fait par
☑ `int* p;` ☐ `int p*;` ☐ `int[] p;` ☐ `int p[];` ☐ `int& p;` ☐ `int p&;`

▶ **Q. 5** Récupérer l'adresse d'une variable `n` se fait par
☐ `*n;` ☐ `n*;` ☐ `[]n;` ☐ `n[];` ☑ `&n;` ☐ `n&;`

▶ **Q. 6** La déclaration d'une fonction `f` prenant un tableau `t` d'entiers en paramètre se fait par
☑ `f(int* t)` ☐ `f(int t*)` ☐ `f(int[] t);` ☑ `f(int t[]);` ☐ c'est impossible.

▶ **Q. 7** La déclaration d'une fonction `f` retournant un tableau d'entiers se fait par
☐ `int* f()` ☐ `int[] f()` ☐ `int[f()]` ☐ `int(f[]);` ☑ c'est impossible.

▶ **Q. 8** Le type des chaînes de caractères en C est
☐ `char` ☑ `char*` ☑ `char []` ☐ `string` ☐ n'existe pas
Il n'y pas de type, à proprement parlé, pour les chaînes en C, mais comme les chaînes sont représentées par des tableaux de caractères avec un caractère `'\0'` à la fin, on utilise les types `char*` ou `char []`.

▶ **Q. 9** Quelle syntaxe permet d'accéder à l'objet pointé par un pointeur `p` et de le modifier ?
☑ `*p = a` ☐ `p = a` ☐ `&p = a` ☐ c'est impossible

▶ **Q. 10** Quelle syntaxe permet de modifier un pointeur `p` ?
☐ `*p = a` ☑ `p = a` ☐ `&p = a` ☐ c'est impossible

▶ **Q. 11** La syntaxe `p->f()` est équivalente à
☐ `p[f()]` ☐ `*(p.f())` ☐ `*p.f()` ☑ `(*p).f()` ☐ n'existe pas

▶ **Q. 12** La syntaxe `p[n]` est équivalente à
☐ `p+n` ☑ `*(p+n)` ☐ `*p+n` ☐ `p*n` ☐ n'existe pas

▶ **Q. 13** Après la déclaration `int n = 0`, la valeur de l'expression `n = n` est
☐ `true` ☐ `false` ☑ `0` ☐ `1` ☐ `2` ☐ aucune ☐ ne compile pas
La valeur d'une affectation est la valeur affectée.

▶ **Q. 14** Après la déclaration `int n = 0`, la valeur de l'expression `n += n` est
☐ `true` ☐ `false` ☑ `0` ☐ `1` ☐ `2` ☐ aucune ☐ ne compile pas

▶ **Q. 15** Après la déclaration `int n = 0`, la valeur de l'expression `n <= n` est
☐ `true` ☐ `false` ☐ `0` ☑ `1` ☐ `2` ☐ aucune ☐ ne compile pas
Il s'agit d'une comparaison avec l'opérateur d'ordre non strict `<=`. La valeur est donc `1` qui représente `true` en C.

▶ **Q. 16** Après la déclaration `int n = 0`, la valeur de l'expression `n == n` est
☐ `true` ☐ `false` ☐ `0` ☑ `1` ☐ `2` ☐ aucune ☐ ne compile pas
Il s'agit d'une comparaison avec l'opérateur d'égalité `==`. La valeur est donc `1` qui représente `true` en C.

▶ **Q. 17** Après la déclaration `int n = 0`, la valeur de l'expression `n != n` est
☐ `true` ☐ `false` ☑ `0` ☐ `1` ☐ `2` ☐ aucune ☐ ne compile pas
Il s'agit d'une comparaison avec l'opérateur d'inégalité `!=`. La valeur est donc `0` qui représente `false` en C.

▶ **Q. 18** Après l'exécution du fragment de code `int n = 0; n = !n;`, la variable `n` a la valeur
☐ `0` ☑ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas
L'opérateur `!` est la négation logique. Il transforme `0` en `1` et inversement car `0` et `1` représentent `false` et `true` en C.

▶ **Q. 19** Après l'exécution du fragment de code `int n = 0; n = n++;`, la variable `n` a la valeur
☑ `0` ☐ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas
L'expression `n++` donne la valeur de `n` avant de l'avoir incrémenté. C'est donc la même valeur qui est ensuite affectée à `n` par l'opérateur `=`.

▶ **Q. 20** Après l'exécution du fragment de code `int n = 0; n = ++n;`, la variable `n` a la valeur
☐ `0` ☑ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas
L'expression `++n` donne la valeur de `n` après l'avoir incrémenté. Cette valeur est ensuite affectée à `n` par l'opérateur `=`.

▶ **Q. 21** Après l'exécution du fragment de code `int n = 1; int* p = &n; p++;`, la variable `n` a la valeur
☐ `0` ☑ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas
L'instruction `p++` incrémente `p` et laisse inchangée la valeur de `n`.

▶ **Q. 22** Après l'exécution du fragment de code `int m = 1; int n = 1; int* p = &m; int* q = &n; n = p == q;`, la variable `n` a la valeur
☑ `0` ☐ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas
Après les initialisations, les pointeurs `p` et `q` contiennent les adresses des variables `m` et `n`. L'expression `p == q` teste si ces deux adresses sont égales et retourne donc `0` puisqu'il s'agit de variables différentes.

▶ **Q. 23** Après l'exécution du fragment de code `int m = 1; int n = 1; int* p = &m; int* q = &n; n = *p == *q;`, la variable `n` a la valeur
☐ `0` ☑ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas
Après les initialisations, les pointeurs `p` et `q` contiennent les adresses des variables `m` et `n`. Les expressions `*p` et `*q` désignent `m` et `n`. L'expression `*p == *q` teste si les valeurs de ces deux variables sont égales et retourne donc `1`.

▶ **Q. 24** Après l'exécution du fragment de code `int m = 1; int n = 1; int* p = &m; int* q = &n; n = &p == &q;`, la variable `n` a la valeur
☑ `0` ☐ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas
L'expression `&p == &q` teste si les adresses des deux pointeurs `p` et `q` sont égales et retourne donc `0` puisqu'il s'agit de deux variables différentes.

► **Q. 25** Après l'exécution du fragment de code `int n = 1; int* p = &n; *p++;`, la variable `n` a la valeur

☐ `0` ☑ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

Après les initialisations, le pointeur `p` contient l'adresse de `n` et `*p` désigne l'emplacement de `n`. L'instruction `*p++` est implicitement parenthésée `*(p++)`. Elle incrémente `p` et retourne la valeur `*p`, c'est-à-dire la valeur de `n` qui reste inchangée.

► **Q. 26** Après l'exécution du fragment de code `int n = 1; int* p = &n; ++*p;`, la variable `n` a la valeur

☐ `0` ☐ `1` ☑ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

Le parenthésage implicite de l'expression est `++(*p)`. Après les initialisations, le pointeur `p` contient l'adresse de `n` et `*p` désigne l'emplacement de `n`. L'expression incrémente `n` qui prend la valeur `2`.

► **Q. 27** Après l'exécution du fragment de code `int n = 1; int* p = &(n+n); n = *p;`, la variable `n` a la valeur

☐ `0` ☐ `1` ☐ `2` ☐ `3` ☐ imprévisible ☑ ne compile pas

L'opérateur `&` ne peut s'appliquer qu'à un emplacement et donc pas à une expression comme `n+n` qui désigne une valeur mais pas un emplacement.

► **Q. 28** Après l'exécution du fragment de code `int n = 1; int* p = &n; n = *p+1;`, la variable `n` a la valeur

☐ `0` ☐ `1` ☑ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

Le parenthésage implicite de l'expression est `n = (*p)+1`. Après les initialisations, le pointeur `p` contient l'adresse de `n` et `*p` désigne l'emplacement de `n`. L'expression affecte à la variable `n` la valeur `2`.

► **Q. 29** Après l'exécution du fragment de code `int n = 1; int* p = &n; n = *(p+1);`, la variable `n` a la valeur

☐ `0` ☐ `1` ☐ `2` ☐ `3` ☑ imprévisible ☐ ne compile pas

`p+1` est l'adresse de l'emplacement qui suit celui de la variable `n`. L'utilisation de cet emplacement dépend des autres variables. L'expression affecte à `n` la valeur contenue dans cet emplacement en l'interprétant comme un entier.

► **Q. 30** Après l'exécution du fragment de code `int n = 1; n++++;`, la variable `n` a la valeur

☐ `0` ☐ `1` ☐ `2` ☐ `3` ☐ imprévisible ☑ ne compile pas

L'expression `n++` désigne une valeur et pas un emplacement. L'opérateur `++` ne peut donc pas être appliqué.

► **Q. 31** Après l'exécution du fragment de code `int n = 1; int* p = &n; int** q = &p; ++**q;`, la variable `n` a la valeur

☐ `0` ☐ `1` ☑ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

Le parenthésage implicite de l'expression est `++(*(*q))`. Après les initialisations, le pointeur `p` contient l'adresse de `n` et le pointeur `q` l'adresse de `p`. Les expressions `*q` et `**q` désignent respectivement `p` et `n`. L'instruction `++**p;` incrémente `n`.

► **Q. 32** Après l'exécution du fragment de code `int n = 0; int m = 1; n = m = n;`, la variable `n` a la valeur

☑ `0` ☐ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

Le parenthésage implicite de l'expression est `n = (m = n)`. L'expression `m = n` affecte la valeur `0` à la variable `m` et donne la valeur `0`. L'expression globale affecte alors la valeur `0` à la variable `n`.

► **Q. 33** Après l'exécution du fragment de code `int n = 0; n = n == n;`, la variable `n` a la valeur

☐ `0` ☑ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

Le parenthésage implicite de l'expression est `n = (n == n)`. L'expression `n == n` est un test d'égalité et donne la valeur `1` qui représente la valeur `true` en C. L'expression globale affecte alors la valeur `1` à la variable `n`.

► **Q. 34** Après l'exécution du fragment de code `int n = 0; n = n++ + n++;`, la variable `n` a la valeur

☐ `0` ☑ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

Quel que soit l'ordre d'évaluation des deux expressions `n++` le résultat est `1`. La première expression `n++` évaluée donne `0` et la seconde `1` et la somme est toujours `1`.

► **Q. 35** Lors de l'exécution du fragment de code `int n = 1; f(n++, n++);`, la fonction `f` est appelée avec les valeurs

☐ `1,1` ☐ `1,2` ☐ `2,1` ☐ `2,2` ☑ imprévisible ☐ ne compile pas

La norme du C++ n'impose pas l'ordre d'évaluation des paramètres d'un appel de fonction et laisse le compilateur choisir. Si le premier paramètre est évalué en premier, la fonction `f` reçoit `1,2` et s'il est évalué en second, la fonction `f` reçoit `2,1`.

► **Q. 36** La fonction `f` est définie par `int f(int n){ return n+1; }`. Après l'exécution du fragment de code `int n = f(1);`, la variable `n` a la valeur

☐ `0` ☐ `1` ☑ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

► **Q. 37** La fonction `f` est définie par `int f(int* p){ return *p+1; }`. Après l'exécution du fragment de code `int n = 1; n = f(n);`, la variable `n` a la valeur

☐ `0` ☐ `1` ☐ `2` ☐ `3` ☐ imprévisible ☑ ne compile pas

La fonction `f` prend une adresse en paramètre alors que l'appel à `f` est fait avec une valeur entière.

► **Q. 38** La fonction `f` est définie par `int f(int* p){ return *p+1; }`. Après l'exécution du fragment de code `int n = 1; n = f(&n);`, la variable `n` a la valeur

☐ `0` ☐ `1` ☑ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

► **Q. 39** La fonction `f` est définie par `int f(int* p){ return *p+1; }`. Après l'exécution du fragment de code `int n = f(&1);`, la variable `n` a la valeur

☐ `0` ☐ `1` ☐ `2` ☐ `3` ☐ imprévisible ☑ ne compile pas

L'opérateur `&` ne peut s'appliquer qu'à un emplacement et donc pas à une constante comme `1`.

► **Q. 40** La fonction `f` est définie par `int* f(){ int n = 1; return &n; }`. Après l'exécution du fragment de code `int n = *f();`, la variable `n` a la valeur

☐ `0` ☐ `1` ☐ `2` ☐ `3` ☑ imprévisible ☐ ne compile pas

La fonction `f` retourne l'adresse de sa variable locale `n` dont l'emplacement est libéré au retour de la fonction.

► **Q. 41** La fonction `f` est définie par `int* f(int* p){ int n = 1; return p; }`. Après l'exécution du fragment de code `int n = 2; n = *f(&n);`, la variable `n` a la valeur

☐ `0` ☐ `1` ☑ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

► **Q. 42** La fonction `f` est définie par `void f(int* p, int* q){ int t = *p; *p = *q; *q = t; }`. Quels sont les appels à la fonction `f` qui échangent les valeurs des variables `m` et `n`.

☐ `f(m, n)` ☐ `f(*m, *n)` ☑ `f(&m, &n)` ☐ `f(!m, !n)` ☐ `f(m+n, m-n)`

Il faut passer en paramètre à la fonction `f` les adresses des variables `m` et `n`.

La fonction `len` est définie de la façon suivante.

```
int len(char* s) {
  char* t = s;
  while(*s++ != '\0');
  return s-t;
}
```

► **Q. 43** Dans le fragment de code `len("abc");`, la fonction `len` retourne la valeur
☐ `-1` ☐ `0` ☐ `3` ☑ `4` ☐ imprévisible ☐ ne compile pas
La fonction `len` incrémente le pointeur `s` à chaque caractère lu, y compris le caractère `'\0'` de fin de chaîne. Elle retourne donc la longueur de la chaîne incrémentée de `1` pour ce caractère.

► **Q. 44** Dans le fragment de code `char* s = "abc"; len(s);`, la fonction `len` retourne la valeur
☐ `-1` ☐ `0` ☐ `3` ☑ `4` ☐ imprévisible ☐ ne compile pas

► **Q. 45** Dans le fragment de code `char s[] = {'a', 'b', 'c'}; len(s);`, la fonction `len` retourne la valeur
☐ `-1` ☐ `0` ☐ `3` ☐ `4` ☑ imprévisible ☐ ne compile pas
Le tableau `s` contient une suite de caractères non terminée par le caractère `'\0'`. Il n'est pas possible de prévoir où va s'arrêter le parcours de la fonction `len`.

► **Q. 46** Après la déclaration `int t[] = {1, 2, 3};`, le tableau `t` contient les valeurs
☐ `0, 0, 0` ☑ `1, 2, 3` ☐ imprévisible ☐ ne compile pas

► **Q. 47** Après la déclaration `int[] t = {1, 2, 3};`, le tableau `t` contient les valeurs
☐ `0, 0, 0` ☐ `1, 2, 3` ☐ imprévisible ☑ ne compile pas
Il s'agit de la syntaxe de Java pour les tableaux qui n'est pas valide en C.

► **Q. 48** Après la déclaration `int* t = {1, 2, 3};`, le tableau `t` contient les valeurs
☐ `0, 0, 0` ☐ `1, 2, 3` ☐ `3, 2, 1` ☐ imprévisible ☑ ne compile pas
La déclaration `int* t;` déclare un pointeur qui ne peut être initialisé avec une suite de valeurs entières. La variable `t` ne peut contenir qu'une adresse.

► **Q. 49** Après l'exécution du fragment `int t[] = {1, 2, 3}; t = t;`, le tableau `t` contient les valeurs
☐ `0, 0, 0` ☐ `1, 2, 3` ☐ imprévisible ☑ ne compile pas
Il n'y a pas d'affectation entre tableaux en C. L'instruction `t = t;` n'est pas valide.

► **Q. 50** Après l'exécution du fragment `int t[] = {1, 2, 3}; int n = &t == t;`, la variable `n` a la valeur
☐ `true` ☐ `false` ☐ `0` ☑ `1` ☐ imprévisible ☐ ne compile pas
Par convention, l'adresse d'un tableau est le tableau lui-même.

► **Q. 51** Après l'exécution du fragment `int s[] = {1, 2, 3}; int t[] = s;`, le tableau `t` contient les valeurs
☐ `0, 0, 0` ☐ `1, 2, 3` ☐ imprévisible ☑ ne compile pas
Il n'est pas possible, en C, d'initialiser un tableau avec un autre tableau.

► **Q. 52** Après l'exécution du fragment `int s[] = {1, 2, 3}; int t[] = {1, 2, 3}; int n = s == t;`, la variable `n` a la valeur
☑ `0` ☐ `1` ☐ `2` ☐ `6` ☐ imprévisible ☐ ne compile pas
L'expression `s == t` teste si les deux tableaux se trouvent à la même adresse et retourne donc la valeur `0`.

► **Q. 53** Après l'exécution du fragment `int t[] = {1, 2, 3}; *t = 3;`, le tableau `t` contient les valeurs

☐ `3, 3, 3` ☑ `3, 2, 3` ☐ `1, 3, 3` ☐ imprévisible ☐ ne compile pas

L'expression `*t` est équivalent à `t[0]` et désigne la première case du tableau `t`.

► **Q. 54** Après l'exécution du fragment `int t[] = {1, 2, 3}; int* p = t; p[1] = 3`, le tableau `t` contient les valeurs

☐ `3, 3, 3` ☐ `3, 2, 3` ☑ `1, 3, 3` ☐ imprévisible ☐ ne compile pas

Après les initialisations, le pointeur `p` contient l'adresse de la première case du tableau `t`. Comme les cases des tableaux sont numérotées à partir de `0`, `p[1]` dédigne la seconde case du tableau `t`.

► **Q. 55** Après l'exécution du fragment de code `int t[] = {1, 2, 3};`
`for(int i = 1; i < 3; i++) t[i] += t[i-1];` le tableau `t` contient les valeurs

☐ `1, 2, 3` ☑ `1, 3, 6` ☐ `1, 1, 1` ☐ imprévisible ☐ ne compile pas

La boucle `for` effectue les deux affectations `t[1] = t[1] + t[0]` puis `t[2] = t[2] + t[1]`.

Le tableau `t` et la fonction `sum` sont définis de la façon suivante.

```
int t[] = { 1, 2, 3, 4, 5 };
int sum(int t[], int n) {
  int s = 0;
  for (int i = 0; i < n; ++i) s += t[i];
  return s;
}
```

► **Q. 56** Quels sont les appels à `sum` qui calculent la somme des quatre premières valeurs du tableau `t` et retournent la valeur `10` ?

☑ `sum(t, 4)` ☐ `sum(t[1], 4)` ☐ `sum(t[0], 4)`
☐ `sum(&t, 4)` ☑ `sum(&t[0], 4)` ☐ `sum(*t, 4)`

Il faut passer en paramètre à la fonction `sum` l'adresse de la première case du tableau qui est donnée par les expressions `t` et `&t[0]`.

► **Q. 57** Quels sont les appels à `sum` qui calculent la somme des quatre dernières valeurs du tableau `t` et retournent la valeur `14` ?

☑ `sum(t+1, 4)` ☐ `sum(&t+1, 4)` ☐ `sum(*(t+1), 4)`
☐ `sum(t[1], 4)` ☑ `sum(&t[1], 4)` ☐ `sum(*t+1, 4)`

Il faut passer en paramètre à la fonction `sum` l'adresse de la deuxième case du tableau qui est donnée par les expressions `t+1` et `&t[1]`.

Le type `P` est défini de la façon suivante.

```
typedef struct {
  int x;
  int y;
} P;
```

► **Q. 58** La déclaration d'une fonction `f` prenant en paramètre une valeur de type `P` se fait par

☑ `f(P p)` ☐ `f(P* p)` ☐ `f(P& p)` ☐ `f(P p[])` ☐ c'est impossible.

► **Q. 59** La déclaration d'une fonction `f` retournant une valeur de type `P` se fait par

☑ `P f()` ☐ `P* f()` ☐ `P& f()` ☐ `P[] f()` ☐ c'est impossible.

► **Q. 60** Après l'exécution du fragment de code `P p = {1, 2}; P q = p; int n = q.y`, la variable `n` a la valeur

☐ `0` ☐ `1` ☑ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

► **Q. 61** Après l'exécution du fragment de code `P p = {1, 2}; int* q = &p.x; int n = *q;`, la variable `n` a la valeur

☐ `0` ☑ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

Après les initialisations, le pointeur `q` contient l'adresse de l'attribut `x` de la structure `p`. La variable `n` reçoit donc la valeur `1`.

► **Q. 62** Après l'exécution du fragment de code `P p = {1, 2}; P* q = &p; int n = q->x;`, la variable `n` a la valeur

☐ `0` ☑ `1` ☐ `2` ☐ `3` ☐ imprévisible ☐ ne compile pas

Après les initialisations, le pointeur `q` contient l'adresse de la structure `p`. L'expression `q->x` qui est équivalente à `(*q).x` désigne donc l'attribut `x` de la structure `p`. La variable `n` reçoit donc la valeur `1`.

► **Q. 63** La déclaration d'une fonction `F` qui prend en paramètre une fonction `f` (prenant en paramètre un entier et retournant un entier) et qui retourne un entier se fait par

☑ `int F(int f(int))` ☐ `int F(int *f(int))` ☑ `int F(int (*f)(int))`

☐ `int F(int f, int n)` ☐ c'est impossible.

► **Q. 64** Après les deux déclarations `typedef int (*fun)(int); fun F(fun f) {return f;}`, l'expression `F(f)(2)` est valide si `f` est défini par

☑ `int f(int n) { return n; }`

☐ `int f(int* p) { return *p; }`

☐ `int* f(int* p) { return p; }`

☐ `int* f(int n) { return &n; }`

☐ `int f() { return 0; }`

☐ ne compile pas