Exercice 1 : Opérateurs combinés

En reprenant à chaque fois les valeurs suivantes, calculer les valeurs de `i`, `j` et lorsque cela s'applique `z` après l'exécution des instructions suivantes :

```
int i = 1, j = 3;
int z;
```

Expression	i	j	z
`i += j`	**4**	**3**	**?**
`i += -j`	**-2**	**3**	**?**
`i -= j`	**-2**	**3**	**?**
`i -= -j`	**4**	**3**	**?**
`i *= j`	**3**	**3**	**?**
`i *= -j`	**-3**	**3**	**?**
`i /= j`	**0**	**3**	**?**
`z = i * j == 6`	**1**	**3**	**0**
`z = i++ * j == 6`	**2**	**3**	**0**
`z = ++i * j == 6`	**2**	**3**	**1**

Exercice 2 : Opérateur ternaire

(a) Simplifiez l'expression suivante.

```
z = (a > b ? a : b) + (a <= b ? a : b) ;
```

```
z = a + b;
```

(b) Soit variable `n` est de type `int`. Écrire une expression unique qui prend la valeur :

1. `-1` si `n` est négatif
2. `0` si `n` est nul
3. `1` si `n` est positif

```
n < 0 ? -1 : (n == 0) ? 0 : 1
```

Exercice 3 : Opérateurs incorrects

Soit les déclarations suivantes, indiquez pourquoi les propositions suivantes sont incorrectes :

```
double f, g = 7;
int i, j = j;
```

(a)

```
int(f) + 1.9
```

> La coercition de type s'écrit `(int)f` et non `int(f)`.

(b)

```
i = 1 + j = j / 2
```

> L'opérateur `=` est moins prioritaire que `+`. En conséquence, cette expression tente d'affecter `j / 2` à `1 + j`. Or, une affectation n'est possible que si l'expression à gauche est une variable, ce qui n'est pas le cas ici.

(c)

```
f = g << 2
```

> Le décalage de bits est un opérateur de type `int`, et non un opérateur de type `double`.

(d)

```
i = ++j++
```

> Cette expression essaie d'exécuter l'opérateur de post-incrémentation à `(++j)` or cette expression est évalué comme une valeur, et non une variable.

(e)

```
i++ = ++j
```

> La partie gauche de l'opérateur `i++` est évaluée comme une valeur, et n'est donc pas assignable. On dit que le membre de gauche n'est pas une *lvalue*.

Exercice 4

Indiquez pour chaque groupe d'instruction ci-dessous si l'expression est correcte ou non. Sinon, expliquer pourquoi.

```
int i;
assert(scanf("%d", &i) == 1);
```

(a)

```
if (!(i < 8) && !(i > 8)) then printf("i vaut 8\n");
```

> Incorrect : une erreur apparaît à la compilation, le mot *then* n'est pas valide en C.

(b)

```
if (!(i < 8) && !(i > 8)) printf("i "); printf("vaut 8\n");
```

> Incorrect : la première instruction est correctement exécutée mais la seconde s'exécute inconditionnellement.

(c)

```
if !(i < 8) && !(i > 8) printf("i vaut 8\n");
```

> Incorrect : l'expression est mal formée, la condition d'embranchement après le *if* doit être entre parenthèses.

(d)
```
if (!(i < 8) && !(i > 8)) printf("i vaut 8\n");
```

Correct !

(e)
```
if (i = 8) printf("i vaut 8\n");
```

Incorrect : affiche que `i` vaut 8 dans tous les cas.

(f)
```
if (i & (1 << 3)) printf("i vaut 8\n");
```

Correct !

(g)
```
if (i ^ 8) printf("i vaut 8\n");
```

Incorrect : affiche *i vaut 8* pour tous les cas sauf lorsque `i` vaut 8 !

(h)
```
if (i - 8) printf("i vaut 8\n");
```

Incorrect : affiche *i vaut 8* pour tous les cas sauf lorsque `i` vaut 8 !

(i)
```
if (i == 1 << 3) printf ("i vaut 8\n");
```

Correct !

(j)
```
if (!((i < 8) || (i > 8))) printf("i vaut 8\n");
```

Correct !

Exercice 5 : Analyse de code

Que voyez-vous sur la sortie standard ?

Notez que selon le standard ISO8899, une expression comportant plusieurs post ou pré incrémentation est indéterminée, néanmoins la logique de l'expression est définie dans la plupart des compilateurs et elle suit la règle enseignée en cours.

```
#include <stdio.h>
int main() {
    int x = 2;
    int y = ++x * ++x;
    printf("%d%d", x, y);
    x = 2;
    y = x++ * ++x;
    printf("%d%d", x, y);
}
```

41648